Demystifying security and compatibility issues in Android Apps



MONASH University

Xiaoyu Sun

Faculty of Information Technology

Monash University

A thesis submitted for the degree of

Doctor of Philosophy

January 2023

Copyright

©2022 - Xiaoyu Sun

Copyright Notice *I certify that I have made all reasonable efforts to secure copyright permissions for third-party content included in this thesis and have not knowingly added copyright content to my work without the owner's permission.*

Abstract

Never before has any OS been so popular as Android. Existing mobile phones are not simply devices for making phone calls and receiving SMS messages, but powerful communication and entertainment platforms for web surfing, social networking, etc. Even though the Android OS offers powerful communication and application execution capabilities, it is riddled with defects (e.g., security risks, and compatibility issues), new vulnerabilities come to light daily, and bugs cost the economy tens of billions of dollars annually. For example, malicious apps (e.g., back-doors, fraud apps, ransomware, spyware, etc.) are reported [Google, 2022] to exhibit malicious behaviours, including privacy stealing, unwanted programs installed, etc. To counteract these threats, many works have been proposed that rely on static analysis techniques to detect such issues. However, static techniques are not sufficient on their own to detect such defects precisely. This will likely yield false positive results as static analysis has to make some trade-offs when handling complicated cases (e.g., object-sensitive vs. object-insensitive). In addition, static analysis techniques will also likely suffer from soundness issues because some complicated features (e.g., reflection, obfuscation, and hardening) are difficult to be handled [Sun et al., 2021b, Samhi et al., 2022].

My research during my Ph.D. is among the first to tackle these problems directly: I develop several tools that attempt to automatically integrate static program analysis techniques with dynamic testing techniques for detecting Android defects in real-world applications. I have demonstrated that my approaches are more effective and precise in detecting more security issues than state-of-the-art program analysis tools. Specifically, my works highlight several security/compatibility aspects of the Android framework and unveil the

discovery of various security/compatibility threats that were never been investigated before. I present three different approaches to solve these issues:

- **SEEKER** – While extremely valuable to achieve many advanced functions, mobile phone sensors can be abused by attackers to implement malicious activities in Android apps, as experimentally demonstrated by many state-of-the-art studies. There is hence a strong need to regulate the usage of mobile sensors so as to keep them from being exploited by malicious attackers. However, despite the fact that various efforts have been put in achieving this, i.e., detecting privacy leaks in Android apps, I have not yet found approaches to automatically detect sensor leaks in Android apps. To fill the gap, I designed and implemented a novel prototype tool, SEEKER, that extends the famous FlowDroid [Arzt et al., 2014] tool to detect sensor-based data leaks in Android apps. SEEKER conducts sensor-focused static taint analyses directly on the Android apps' bytecode and reports not only sensor-triggered privacy leaks but also the sensor types involved in the leaks. Experimental results using over 40,000 real-world Android apps show that SEEKER is effective in detecting 1,964 sensor leaks in Android apps, and malicious apps are more interested in leaking sensor data than benign apps. Moreover, our improvement for supporting field sources detection has been merged to the original version FlowDroid via pull #385 on GitHub[Xiaoyu Sun, 2021]), which I believe could be adapted to analyze other sensitive field-triggered leaks.

- **HiSenDroid –** Security of Android devices is now paramount, given their wide adoption among consumers for many sensitive tasks and data management. As researchers develop tools for statically or dynamically detecting suspicious apps, malware writers regularly update their attack mechanisms to hide malicious behavior implementation. This poses two problems to current research techniques: static analysis approaches, given their over-approximations, can report an overwhelming

number of false alarms, while dynamic approaches will miss those behaviors that are hidden through evasion techniques. I propose in this work a static approach specifically targeted at highlighting hidden sensitive operations, mainly sensitive data flows. The prototype version of HiSenDroid has been evaluated on a large-scale dataset of thousands of malware and goodware samples on which it successfully revealed anti-analysis code snippets aiming at evading detection by dynamic analysis. I further experimentally show that, with FlowDroid, some of the hidden sensitive behaviors would eventually lead to private data leaks. Those leaks would have been hard to spot either manually among the large number of false positives reported by the state of the art static analyzers, or by dynamic tools. Overall, by putting the light on hidden sensitive operations, HiSenDroid helps security analysts in validating potential sensitive data operations, which would be previously unnoticed.

- **JUNITTESTGEN** – Despite being one of the largest and most popular projects, the official Android framework has only provided test cases for less than 30% of its APIs. Such a poor test case coverage rate has led to many compatibility issues that can cause apps to crash at runtime on specific Android devices, resulting in poor user experiences for both apps and the Android ecosystem. To mitigate this impact, various approaches have been proposed to automatically detect such compatibility issues. Unfortunately, these approaches have only focused on detecting signature-induced compatibility issues (i.e., a certain API does not exist in certain Android versions), leaving other equally important types of compatibility issues unresolved. In this work, I propose a novel prototype tool, JUnitTestGen, to fill this gap by mining existing Android API usage to generate unit test cases. After locating Android API usage in given real-world Android apps, JUnitTestGen performs inter-procedural backward data-flow analysis to generate a minimal executable code snippet (i.e., test case). Experimental results on thousands of real-world Android apps show that JU-

nitTestGen is effective in generating valid unit test cases for Android APIs. I show that these generated test cases are indeed helpful for pinpointing compatibility issues, including ones involving semantic code changes.

Thesis including published works declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

This thesis includes 3 original papers published in peer-reviewed journals/conferences. The core theme of the thesis is demystifying security and compatibility issues in Android Apps. The ideas, development and writing up of all the papers in the thesis were the principal responsibility of myself, the student, working within the Faculty of Information Technology of Monash University under the supervision of John Grundy and Li Li.

The inclusion of co-authors reflects the fact that the work came from active collaboration between researchers and acknowledges input into team-based research. In the case of Chapter 3,4,5, my contribution to the work involved the following:

Thesis Chapter	**Publication Title**	**Status** *(published, in press, accepted or returned for revision, submitted)*	**Nature and % of student contribution**	**Co-author name(s) Nature and % of Co-author's contribution***	**Co-author (s), Monash student Y/N***
3	Characterizing Sensor Leaks in Android Apps	*Accepted*	*55%. All of the experiment, writing the manuscript and revisions for the paper*	1)Xiao Chen. *input into manuscript 10%* 2) Kui Liu. *input into manuscript and discuss with key challenges, 10%* 3) Sheng Wen. *input into manuscript 5%* 4) Li Li. *Concept, 10% and discuss with key challenges* 5) John Grundy. *input into manuscript and discuss with key challenges, 10%*	*None of them*

4	Demystifying Hidden Sensitive Operations in Android apps	*Accepted*	*50%. Concept and collecting data and doing experiments and writing first draft*	1)Xiao Chen. *input into manuscript 10%* 2) Li Li. *Concept, input into manuscript and discuss with key Challenges, 10%* 3) Haipeng Cai. *input into manuscript 5%* 4) John Grundy. *input into manuscript and discuss with key challenges, 10%* 5) Jordan Samhi. *input into manuscript 5%* 6) Tegawendé F. Bissyandé. *input into manuscript 5%* 7) Jacques Klein. *input into manuscript 5%*	*None of them*
5	Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues	*Accepted*	*60%. Concept and collecting data and doing experiments and writing first draft*	1)Xiao Chen. *input into manuscript and discuss with key Challenges, 10%* 2) Yanjie Zhao. *input into manuscript 5%* 3) Pei Liu. *input into manuscript 5%* 4) Li Li. *Concept, input into manuscript and discuss with key Challenges, 10%* 5) John Grundy. *input into manuscript and discuss with key Challenges, 10%*	1.*No* *2.Yes* *3.Yes* *4.No* *5.No*

I have renumbered sections of published papers in order to generate a consistent presentation within the thesis.

Student name: Xiaoyu Sun

Student signature: Xiaoyu Sun **Date: 27-Jan-2023**

The undersigned hereby certify that the above declaration correctly reflects the nature and extent of the student's and co-authors' contributions to this work. In instances where I am not the responsible author I have consulted with the responsible author to agree on the respective contributions of the authors.

Student name: John Grundy

Student signature: John Grundy **Date: 27-Jan-2023**

Publications

Publications included in this thesis

1. **Xiaoyu Sun**, Xiao Chen, Kui Liu, Sheng Wen, Li Li, and John Grundy. 2021. Characterizing Sensor Leaks in Android Apps. The IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE) 2021, 12 pages. (Chapter 3)

2. **Xiaoyu Sun**, Xiao Chen, Li Li, Haipeng Cai, John Grundy, Jordan Samhi, Tegawendé F. Bissyandé and Jacques Klein, 2022, Demystifying Hidden Sensitive Operations in Android apps. ACM Transactions on Software Engineering and Methodology (TOSEM) 2022, 30 pages. (Chapter 4)

3. **Xiaoyu Sun**, Xiao Chen, Yanjie Zhao, Pei Liu, John Grundy and Li Li. 2022. Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues. The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2022, 13 pages. (Chapter 5)

Other publications during candidature

1. **Xiaoyu Sun**, Li Li, Tegawendé F. Bissyandé, Jacques Klein, Damien Octeau, and John Grundy, 2021, Taming reflection: An essential step toward whole-program analysis of android apps. ACM Transactions on Software Engineering and Methodology (TOSEM) 2021, 35 pages.

2. **Xiaoyu Sun**, Xiao Chen, Yonghui Liu, John Grundy, Li Li. 2022. Taming Android Fragmentation through Lightweight Crowdsourced Testing. IEEE Transactions on Software Engineering (TSE) 2023, 18 pages.

Acknowledgements

I am incredibly fortunate to have many people to thank.

First and foremost, I would like to thank my supervisors Li Li and John Grundy, who guided me into the land of program analysis. Their immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life. Every meeting I had with Li and John was fruitful – they could consistently provide insightful suggestions and was also willing to dive deeply into technical discussions. It is their mentorship and encouragement which made this thesis possible.

I would like to thank Michael Sheng, who hosted me during my visit to Macquarie University. Michael is insightful and has the fantastic ability to explain complicated things in an abstract and easy-to-follow manner. I am also very grateful for his long-lasting support with my research career.

Great thanks to Jin-Song Dong, who hosted me during my visit at Griffith University. Dong has lots of exciting ideas, and his research vision has always inspired me. I expect much more collaboration to come. I also thank Zhe Hou and the whole research team for their generous support of this visit.

I would like to thank all the people who contributed in some way to the work described in this thesis. Every result described in this thesis was accomplished with the help and support of fellow labmates and collaborators. Jacques Klein and I worked together on several projects, and without his efforts my job would have undoubtedly been more difficult. I greatly benefited from his keen scientific insight, his knack for solving seemingly intractable practical difficulties, and his ability to put complex ideas into simple terms. Yanjie Zhao joined our research group when I was beginning my first year as a graduate

student, and, with her knowledge of program analysis and relentless work ethic, she was able to successfully carry out the fixable patch for incompatibility in Android. I was fortunate to have the chance to work with Dr. Pei Liu, who worked closely with me during my PhD. Moreover, I would like to thank the various members from the University of Luxembourg with whom I had the opportunity to work and have not already mentioned: Jordan Samhi, Jun Gao and Tegawendé F Bissyandé.

Last but not least, I would like to thank my parents for their unconditional love and support. I made it through thanks to all of you.

Contents

1 Introduction **1**

1.1 Background 1

1.1.1 Android Security and Compatibility Issues 1

1.1.2 Sensor Leak Detection 2

1.1.3 Hidden Sensitive Operation Detection 3

1.1.4 Compatibility Issues Detection 4

1.2 Motivations and Objectives 7

1.2.1 Motivations 7

1.2.2 Research objectives 10

1.3 Thesis Scope 12

1.4 Key Thesis Contributions 16

1.5 Structure of the thesis 18

2 Literature Review **20**

2.1 Static analysis for sensor leak detection 20

2.2 Evasive techniques detection 22

2.3 Compatibility issues detection 25

3 Characterizing Sensor Leaks in Android Apps **28**

3.1 Introduction 28

3.2 Related Work 31

3.2.1 Android sensor usage 31

3.2.2 Software side-channels attacks. 32

3.2.3 Static analysis on Android apps. 32
3.3 Motivation 33
3.4 Approach 35
3.4.1 Sensitive Sensor Source Identification 35
3.4.2 Sensor-triggered Static Taint Analysis 37
3.4.3 Sensor Type Inference 38
3.5 Experimental Setup 40
3.6 Results and Analysis 41
3.6.1 RQ1 – Feasibility of Detecting Sensor Data Leaks 41
3.6.2 RQ2 – Characterization of Sensor Leaks 43
3.6.3 RQ3 – Runtime Overhead 48
3.7 Discussion 49
3.7.1 Implications 49
3.8 Threats to Validity 50
3.9 Summary 52

4 Demystifying Hidden Sensitive Operations in Android apps 53
4.1 Introduction 53
4.2 HSO Definition and Motivation 57
4.3 Approach 59
4.3.1 Hidden Sensitive Branch Location 60
4.3.2 Trigger Condition Inference 61
4.3.3 Suspicious HSO Detection 63
4.4 Conventional Usage 65
4.5 Suspicious HSO Analysis 72
4.5.1 Trigger Conditions 75
4.5.2 Sensitive APIs involved in suspicious HSOs 80

4.5.3 Trigger Condition to Hidden Sensitive API Pairs 80
4.5.4 Suspicious HSOs in Third-party Code 82
4.5.5 Comparison with state-of-the-art 83
4.5.6 Impact of Code Obfuscation 85
4.6 Implication 88
4.7 Limitations 91
4.8 Related Work 94
4.9 Summary 96

5 Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues 98

5.1 Introduction 98
5.2 Motivation 102
5.3 Approach 106
5.3.1 Automated API Unit Test Generation 106
5.3.2 App Compatibility Testing 115
5.4 Evaluation 116
5.4.1 Experimental Setup 116
5.4.2 RQ1-Effectiveness 117
5.4.3 RQ2-Performance on Real-World Applications 119
5.4.4 RQ3-Comparison with State-of-the-art 124
5.5 Discussion 126
5.5.1 Implications 126
5.6 Limitations 127
5.7 Related Work 130
5.8 Summary 131

6 Conclusions and Future Research **132**

6.1 Key Contributions 132

6.2 Future Work 135

6.3 Summary 139

Bibliography **140**

List of Figures

1.1 The snoopy example of sniffing smartwatch passwords via censoring motion sensor data [Lu et al., 2018]. 7

3.1 The working process of our approach. 35
3.2 Distribution of sensor leaks in each app. 42
3.3 Code snippet of sensor value leak excerpted from *com.tencent.pb*. 47
3.4 Distribution of time Performance spent to analyze an app by FlowDroid and SEEKER, respectively. 48

4.1 The working process of HiSenDroid. 60
4.2 The simplified working process of the trigger condition inference module. The code is presented in simplified Jimple, which is an intermediate representation of Soot [Lam et al., 2011]. Soot is the underline static analysis framework leveraged by HiSenDroid to achieve the backward data-flow analysis. 64
4.3 Distribution of the number of HSOs in *benign set* and *malware set*. 67
4.4 Distribution of the number of *suspicious* HSOs in *benign set* and *malware set*. 74
4.5 Categories of Trigger Conditions in HSO and Sensitive APIs in Malware. . 79
4.6 The HSO Trigger-Sensitive API Pairs Graph. 82
4.7 Results of Sensitive data Flows in Android Apps. 91

5.1 The source code of Android API *getNotificationPolicy()*. 103
5.2 The working process of our approach. 107

5.3 Distribution of the number of valid and invalid test cases per APK. 117

5.4 Distribution of time spent by JUnitTestGen to generate test cases per APK. 119

List of Tables

3.1 Examples of Sensor-based Cybersecurity attacks. 34

3.2 The list of sensitive sensor sources. 36

3.3 Experimental results of the detected sensor leaks. 42

3.4 Number of identified method/field-triggered sensor leaks. 43

3.5 Top-10 frequent leaking sources. 44

3.6 Top-10 frequent sensor types of field-triggered sensor leaks. 44

4.1 Number of suspicious HSOs. 73

4.2 Categories of Trigger Conditions in HSO 76

4.3 Details of The Top 10 Classes of Hidden Sensitive APIs in HSO 81

4.4 Top 10 Trigger Condition to Hidden Sensitive API Pairs. 82

4.5 The comparison results between HiSenDroid and TSOpen. 85

4.6 The comparison results of HiSenDroid before and after obfuscation techniques in malware. 88

4.7 The list of selected source methods that, by default, are not included by FlowDroid. 91

5.1 Categories and statistics of the observed error/exception types associated with compatibility issues. 122

5.2 The comparison results between JUnitTestGen and Evosuite, CiD. 124

Chapter 1

Introduction

1.1 Background

1.1.1 Android Security and Compatibility Issues

Android is the most adopted mobile operating system in terms of users, apps and developers [Anthony Scarsella, 2020]. However, its popularity means that good Android apps must co-exist with apps with security defects. Reports on many different kinds of Android defects are presented in the technology and lay media. For example, Trojans are the most prominent mobile threats as they constitute over 95% of mobile malware [Kaspersky-Lab, 2022]. Over 98% of mobile banking attacks target Android devices, which also comes as no surprise [Kaspersky-Lab, 2022]. Also, cyber scams (e.g., phishing and social media scams) are blamed to obtain sensitive personal information (e.g., financial information or account passwords). In fact, during the pandemic, phishing attacks grew by 600% and became the top infection method in 2021 [Google, 2022]. The increasing number of Android security issues in the recent years put both Android users and app developers in danger. So the need for security is greater than ever for revealing these severe security issues.

On the other hand, Android is also known to suffer from fragmentation problems (a.k.a, compatibility issues). At a high level, Android Fragmentation refers to the fact that there are a massive number of different Android OS versions available on the market. Not every Android user would update their particular OS with the pace of Android framework evolution. As reported, Android API evolution is a critical issue in software maintenance [Nagappan and Shihab, 2016, Li et al., 2018a, Martin et al., 2016, Li et al., 2016a, Oliveira

et al., 2018, Lamothe et al., 2020, Li et al., 2018c, Zhou and Walker, 2016, Dig and Johnson, 2005, Kapur et al., 2010, Liu et al., 2022b, Sun et al., 2021a]. McDonnell et al. [McDonnell et al., 2013] have shown that the Android system updates 115 APIs per month on average, while app developers usually adopt the new APIs much more slower. The slow adoption of API updates may raise various issues, such as security and compatibility. An empirical study on StackOverflow conducted by Linares-Vasquez et al. [Linares-Vásquez et al., 2014] suggests that API updates would trigger more discussions, especially if APIs are removed from the Android system. They also revealed that users are in more favour of apps that use less fault and change-prone APIs [Linares-Vásquez et al., 2013, Bavota et al., 2014], as these apps would likely introduce fewer failures, crashes and other bugs.

In this thesis, I wanted to focus on addressing specific types of security and compatibility issues commonly found in Android apps.

1.1.2 Sensor Leak Detection

I found that mobile phone sensors can be abused by attackers to implement malicious activities, as experimentally demonstrated by many state-of-the-art studies [Aviv et al., 2012, Xu et al., 2012, Schlegel et al., 2011] . Hence there is a strong need to regulate the usage of mobile sensors so as to keep them from being exploited by malicious attackers. However, despite the fact that various efforts have been put in achieving this, i.e., detecting privacy leaks in Android apps, I have not yet found approaches to automatically detect sensor leaks in Android apps. To fill the gap, I design and implement a novel prototype tool, SEEKER, that extends the famous FlowDroid tool to detect sensor-based data leaks in Android apps. SEEKER conducts sensor-focused static taint analyses directly on the Android apps' bytecode and reports not only sensor-triggered privacy leaks but also the sensor types involved in the leaks.

1.1.3 Hidden Sensitive Operation Detection

Other malicious behaviours I wanted to investigate relate to 'sensitive' operations in apps. I found that malware writers regularly update their attack mechanisms to hide malicious behaviour implementation, in order to bypass static code analysis (e.g., via obfuscation) and even dynamic detection (e.g., sensing of sandbox execution). In practice, sophisticated code obfuscation techniques [Moser et al., 2007] are being leveraged by attackers to hide their malicious program behavior, leading to false negatives in most static analyses thus resulting in imprecise and unsound results. Camouflage techniques have been frequently leveraged by attackers to evade dynamic testing approaches [Rasthofer et al., 2017, Egele et al., 2008]. Attackers often introduce a so-called logic bomb or time bomb to set off malicious functions only after certain conditions are met. For instance, after knowing that Google applies a dynamic analysis tool called *bouncer* to scan every app submitted to Google Play for five minutes, as revealed by Oberheide et al. [Oberheide and Miller, 2012], a bunch of malicious apps has been created and been demonstrated to be capable of penetrating Google's bouncer vetting system by simply waiting five minutes before triggering their malicious behavior.

To cope with such hidden malicious behaviors, researchers have devised new detection approaches. For example, Fratantonio et al. [Fratantonio et al., 2016] have proposed an approach called TriggerScope to detect hidden behaviors triggered by predefined circumstances such as events related to location, time, and SMS. However, TriggerScope is not capable of detecting such malicious activities hidden behind other trigger types, such as the existence of other services (i.e., other than location, time and SMS). In line with this research, Pan et al. [Pan et al., 2017] have proposed a machine learning-based approach aiming to discover unknown trigger types. Their approach, however, needs to manually label a dataset for training, which is known to be resource-intensive and error-prone.

Static analyzers suffer less than dynamic approaches from evasion techniques such as

logic bomb or time bomb. In particular, regarding sensitive flow detection (also called privacy leak detection), numerous static analysis tools have been proposed such as FlowDroid [Arzt et al., 2014] (and its extension IccTA [Li et al., 2015c]), AMANDROID [Wei et al., 2014], or DroidSafe [Gordon et al., 2015]. Although these tools are able to track sensitive flows (which are often hidden) by bringing key new contributions to the research community, they still face some well-known limitations [Qiu et al., 2018]: their inherent over-approximations inevitably lead to false alarms, which, for some analyzers, occur at a high rate, making them impractical. Consequently, when building on static analysis, manual investigation is often required. Unfortunately, such efforts cannot scale. Dynamic validation then appears as an alternative. Unfortunately, runtime execution often misses hidden sensitive flows due to the implementation of evasion techniques by attackers. While some effort (e.g., [Fratantonio et al., 2016, Pan et al., 2017]) has been put to *characterize* Hidden Sensitive Operations (HSOs) in Android apps, our community has not yet proposed dedicated approaches to *detect and explain* such operations, allowing attackers to achieve malicious behaviors while bypassing certain security vetting mechanisms.

I fill this research gap by proposing a new prototype tool, HiSenDroid, which deploys an automated static app analyzer tailored for detecting *hidden* sensitive operations.

1.1.4 Compatibility Issues Detection

My focus of research has also included compatibility issues that might also lead to poor user experience. The Android framework provides thousands of public APIs that are heavily leveraged by app developers to facilitate their development of Android apps. Ideally, each such public API should be provided with a set of unit tests to ensure that the API is correctly implemented and the continuous evolution of the framework will not change its semantics. Unfortunately, based on my preliminary investigation, less than 30% of APIs,

are provided with unit test cases[1], leaving the majority of APIs uncovered. This is unacceptable considering that the Android framework nowadays has become one of the most popular projects (with millions of devices running it).

This poor test coverage of Android APIs has led to serious compatibility issues in the Android ecosystem, as recently shown [Li et al., 2018b, Wei et al., 2016, Mutchler et al., 2016, Zhang et al., 2015, Ham and Park, 2011, Huang et al., 2018]. For example, Li et al. [Li et al., 2018b] demonstrate that various Android APIs suffer from compatibility issues as the evolution of the Android framework will regularly remove APIs from or add APIs into the framework. Such API removal or addition can result in no such class or method runtime exceptions when the corresponding app is running on certain framework versions. Liu et al. [Liu et al., 2021a] further present an approach to detect silently-evolved Android APIs, which could cause another type of compatibility issue as their semantics are altered (while not explicitly documented) due to the evolution of the Android framework. Moreover, Wei et al. [Wei et al., 2019] experimentally show that some Android APIs could even be customized by smartphone manufacturers, leading to another type of compatibility issue that causes Android apps to crash on certain devices while behaving normally on others. The authors further propose a prototype tool called PIVOT [Wei et al., 2019] to automatically learn device-specific compatibility issues from existing Android apps. Their experiments on a set of top-ranked Google Play apps have discovered 17 device-specific compatibility issues.

To the best of our knowledge, the state-of-the-art works targeting compatibility issue detection leverage static analysis techniques to achieve their purpose. However, as known to the community, the static analysis will likely yield false positive results as it has to make some trade-offs when handling complicated cases (e.g., object-sensitive vs. object-insensitive). In addition, the static analysis will also likely suffer from soundness issues

[1]Here, we downloaded the source code of AOSP from API level 21 to 30 and then calculated the number of public APIs and their corresponding unit test cases in the `platform/frameworks/base` path.

because some complicated features (e.g., reflection, obfuscation, and hardening) are difficult to be handled [Sun et al., 2021b, Samhi et al., 2022]. Furthermore, except for syntactic changes, compatibility issues could also be triggered by semantic changes, which are non-trivial to be handled statically. Indeed, as demonstrated by Liu et al. [Liu et al., 2021a], there are various semantic change-induced compatibility issues in the Android ecosystem that remain undetected after various static compatibility issue detection approaches are proposed to the community.

Moreover, static app analysis can only be leveraged to perform post-momentum analysis (i.e., after the compatibility issues are introduced to the community). They cannot stop the problems from being distributed into the community – many Android apps, including very popular ones, still suffer from compatibility issues. To mitigate this problem, I argued that incompatible Android APIs should be addressed as early as possible, i.e., ideally, at the time when they are introduced to the framework. This could be achieved by providing unit tests for every API introduced to the framework and regressively testing the APIs against Android devices with different manufacturers and different framework versions. However, it is time-consuming to manually write and maintain unit tests for each Android API (which probably explains why there is only a small set of APIs covered by unit tests at the moment). There is hence a need to automatically generate compatibility unit tests for Android APIs.

To adddress this, I develop a prototype tool, JUnitTestGen, that attempts to automatically generate test cases for Android APIs based on their practical usage in real-world apps.

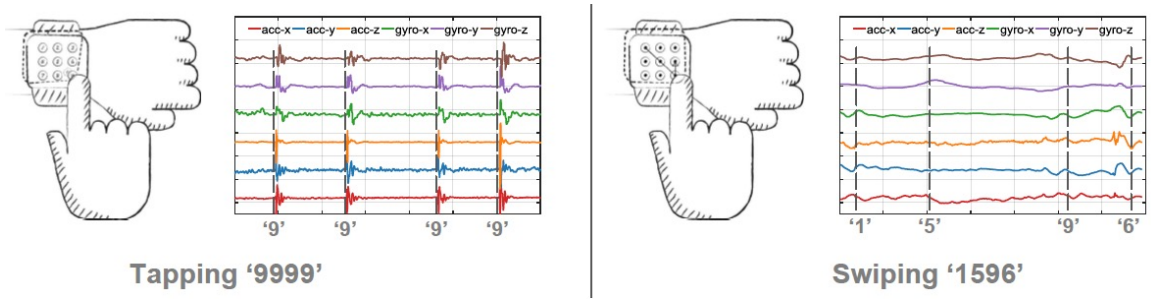


Figure 1.1: The snoopy example of sniffing smartwatch passwords via censoring motion sensor data [Lu et al., 2018].

1.2 Motivations and Objectives

1.2.1 Motivations

To overcome the security and compatibility issues, other researchers have proposed various approaches to mitigate the usage of security/compatibility issues in Android apps [Wei et al., 2016, Wei et al., 2019, Xia et al., 2020, Crandall et al., 2006, Brumley et al., 2008, Zheng et al., 2012]. These approaches mainly leverage static analysis to achieve their purpose. Unfortunately, static analysis is known to likely generate false-positive and false-negative results and is yet hard to handle many issues that involve sophisticated semantic changes [Liu et al., 2021a]. In addition, the static analysis will also likely suffer from soundness issues because some complicated features (e.g., hidden sensitive operations) are difficult to be handled [Sun et al., 2021b, Samhi et al., 2022]. Therefore, there is a imperative need to provide more precise and comprehensive approaches to complement existing works in handling sophisticated security/compatibility issues. Here, I broke down the motivations regarding to the three research projects as follows:

- **Sensor Leaks in Android Apps:**

  As a concrete example, Lu et al. [Lu et al., 2018] revealed that sensitive intercepting password could be accessed through motion data on the smartwatch's onboard sensors. They proposed *Snoopy*, a password extraction and inference approach via

sensor data for PIN attack, which could affect smartwatch users in a non-invasive way. *Snoopy* extracts the segments of motion data when users entered passwords and then applies deep learning techniques to infer the actual passwords. Figure 1.1 gives two examples of the differences of the motion sensor data changes when the user swipes or taps a password on a smartwatch. *Snoopy* demonstrates the feasibility of sensor data leaks by intercepting password information entered on smartwatches. Such real-world sensor-enabled attacks motivated us to provide automatic tools for characterizing universal sensor leaks in Android Apps that have been long overlooked.

- **Hidden Sensitive Operations in Android apps:**

```
1  public class MainActivity extends AppCompatActivity {
2   protected void onCreate(Bundle savedInstanceState) {
3    SmsManager smsManager = SmsManager.getDefault();
4    ED ed = new ED(this);
5    StringBuilder message = new StringBuilder();
6
7    if(ed.checkPackageName()) {
8     TelephonyManager tm = (TelephonyManager)
       getSystemService(Context.TELEPHONY_SERVICE);
9     String imei = tm.getDeviceId();
10    String phoneNumber = tm.getLine1Number();
11    String subscriberId = tm.getSubscriberId();
12    message.append(imei);
13    message.append(phoneNumber);
14    message.append(subscriberId);
15    smsManager.sendDataMessage("+115800763861", null, (short)1001,
       message.toString().getBytes(), null, null);
16   } else {
17    //benign string operations
18 }}
19
20
21 public class ED {
22  public ED(Context pContext) {
23   mContext = pContext;
24   mListPackageName.add("com.google.android...genymotion");
25   mListPackageName.add("com.bluestacks");
26   mListPackageName.add("com.bignox.app");
27  }
28  public boolean checkPackageName() {
29   if (!isCheckPackage || mListPackageName.isEmpty()) {
30    return false;
31   }
32   final PackageManager packageManager = mContext.getPackageManager();
33   for (final String pkgName : mListPackageName) {
34    final Intent tryIntent = packageManager.getLaunchIntentForPackage(pkgName);
35    if (tryIntent != null) {
36     final List<ResolveInfo> resolveInfos =
      packageManager.queryIntentActivities(tryIntent,
      PackageManager.MATCH_DEFAULT_ONLY);
37     if (!resolveInfos.isEmpty()) {
38      return true;
```

```
39        }
40      }
41    }
42    return false;
43  }
```

Listing 1.1: An example of a real-world hidden sensitive data flow.

Listing 1.1 exemplifies a simplified code snippet illustrating these definitions in practice. Note that Listing 1 presents the typical characteristics of an HSO in many real-world apps that I have manually analyzed. At line 7, the app firstly checks if it is running on one of the popular Android emulators (i.e., *genymotion, bluestacks, and bignox*). If not, the app reads the device information and sends it to a hard-coded phone number through an SMS. Otherwise, if an emulator environment is detected, it will only perform some unharmful string operations (ignored). In this example, three private data – namely the device's IMEI, IMSI, and phone number – are retrieved in lines 9-11 and sent to a hard-coded phone number via SMS (line 15). All of these three leaks are hidden behind the trigger condition *ed.checkPackageName()* (line 7). The trigger condition checks the return value of a self-defined method *checkPackageName()* (line 30), which is determined by several other *if-conditions* defined in the invoking method (lines 31,37,39). Finally, the trigger condition in the HSO is traced back to a system API *PackageManager.queryIntentActivities()* (line 38). This trigger condition examines whether popular Android emulator packages (lines 26-28) are available in the device, i.e., checking if the app is running on these emulators. If the running environment is not one of the hard-coded emulators, the HSO will be performed. Otherwise, benign string operations are executed (lines 17-19).

- **Compatibility Issues in Android apps:**

  I conducted a preliminary study investigating the test case coverage in the Android framework. Specifically, I downloaded the source code of AOSP from API level 21

to 30 and then calculated the number of public APIs[2] and their corresponding unit test cases provided by Google. Our results revealed that on average less than 30% of Android framework APIs have provided test cases in each API level, indicating the Android framework has a poor test case coverage. When more APIs are provided with unit test cases, more compatibility issues of APIs will likely be identified during regression testing. This will enable them to be fixed at an earlier stage to avoid the introduction of compatibility issues in the first place. To this end, I proposed to effectively and efficiently detect compatibility issues through a dynamic testing approach that fulfills its objective by automatically generating valid test cases by mining API usages from real-world Android apps.

1.2.2 Research objectives

There still widely exist many undetected security and compatibility issues in Android apps. To tackle these, I developed several automatic tools in detecting these issues to the following several aspects:

1. Sensor leaks in Android apps can be abused by attackers to implement malicious activities, i.e., inferring actual password based on motion sensor data. Also, there is no approaches to automatically detect sensor leaks in Android apps.

2. Malware writers regularly update their attack mechanisms to hide malicious behavior implementation. Such behaviors will only be triggered under special circumstances such as at a specific location or in a certain time period. This poses two problems to current research techniques: static analysis approaches, given their over-approximations, can report an overwhelming number of false alarms, while dynamic approaches will miss those behaviors that are hidden through evasion techniques.

[2]The APIs in `platform/frameworks/base` path.

3. Despite being one of the largest and most popular projects, the official Android framework has only provided test cases for less than 30% of its APIs. Such a poor test case coverage rate has led to many compatibility issues that can cause apps to crash at runtime on specific Android devices, resulting in poor user experiences for both apps and the Android ecosystem.

Three key research objectives are raised regarding the above three reasons as follows:

- **Research Objective 1:** *Characterizing Sensor Leaks in Android Apps. An automatic approach should be proposed to detect sensor leaks in Android apps before publishing them onto app markets. Specifically, I needed to pinpoint privacy leaks that are triggered by sensor-related fields and methods. In addition, the sensor type information should be identified to further help security analysts to understand sensor leaks.*

- **Research Objective 2:** *Demystifying Hidden Sensitive Operations in Android apps. Static code analysis should be applied to uncover hidden sensitive operations that will only be triggered under special circumstances such as at a specific location or in a certain time period. Additionally, I needed to provide details aiming at helping security analysts understand why a given hidden sensitive operation is flagged as such.*

- **Research Objective 3:** *Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues. Firstly, static code analysis techniques should be applied to mine the API usages from real-world apps. Then, after constructing test cases for Android APIs, the dynamic execution of such tests on Android devices can help to pinpoint compatibility issues, including not only signature-based but also semantics-based compatibility issues.*

1.3 Thesis Scope

This thesis focuses on providing more precise and comprehensive results in detecting security/compatibility issues in Android apps. I aim to develop automatic tools that can tame these issues in an automatic way. Considering the state-of-the-art tools are not sufficient to detect all sorts of security/compatibility issues, it is therefore imperative to propose novel techniques to address the aforementioned research objectives respectively:

- **SEEKER** – I leverage static analysis to automatically detect privacy leaks originated from Android sensors – Research Objective 1
- **HISENDROID** – I leverage control flow and data flow analyses to identify the unique code level characteristics of hidden sensitive operations – Research Objective 2
- **JUNITTESTGEN** – I perform field-aware, inter-procedural backward data-flow analysis to automatically generate unit test cases for APIs based on their existing usages – Research Objective 3

SEEKER (Chapter3): Despite being needed to support the implementation of many diverse Android apps, mobile phone sensors can also be abused to achieve malicious behaviors. There have been many reports of apps that exploit sensors in Android devices to conduct malicious activities. For example, sensor data are known to be leaked out because they are not protected by any permissions in Android. There is hence a strong need to regulate the usage of mobile sensors so as to keep them from being exploited by malicious attackers. However, despite the fact that various efforts have been put in achieving this, i.e., detecting privacy leaks in Android apps, I have not yet found approaches to automatically detect sensor leaks in Android apps.

To fill the gap, I have presented a novel tool, SEEKER [Sun et al., 2021a], for characterizing sensor leaks in Android apps. Specifically, I extended the famous FlowDroid [Arzt

et al., 2014] tool to detect sensor-based data leaks in Android apps. SEEKER conducts sensor-focused static taint analyses directly on the Android apps' bytecode and reports not only sensor-triggered privacy leaks but also the sensor types involved in the leaks.

Our experimental results on a large scale of real-world Android apps indicate that our tool is effective in identifying all types of potential sensor leaks in Android apps. Our tool is not only capable of detecting sensor leaks, but also pinpointing general privacy leaks that are triggered by class fields. Although there are related works on sensor usage analysis, to the best of our knowledge, there is no other work that thoroughly analyses Android sensor leakage. Unlike previous works, our tool is the first one to characterize all kinds of sensor leaks in Android apps. I extended FlowDroid for supporting field sources detection (i.e., merged to FlowDroid via pull #385 on Github[Xiaoyu Sun, 2021]), which I believed could be adapted to analyze other sensitive field-triggered leaks. To benefit our fellow researchers and practitioners towards achieving this, I have made our approach open source at the following Github site: `https://github.com/MobileSE/SEEKER`.

- *This work has led to a research paper –* ***Characterizing Sensor Leaks in Android Apps****, which has been published in the IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE) in 2021.*

HISENDROID (Chapter4): The emergence of many different malware detection techniques has stimulated malware attackers into being more innovative to increasingly better hide malicious behaviour, in order to bypass static code analysis (e.g., via obfuscation) and even dynamic detection (e.g., sensing of sandbox execution). In practice, sophisticated code obfuscation techniques [Moser et al., 2007] are being leveraged by attackers to hide their malicious program behavior, leading to false negatives in most static analyses thus resulting in imprecise and unsound results.

To cope with such hidden malicious behaviors, I presented to the community a prototype tool called HiSenDroid, which performs a static code analysis to uncover hidden

sensitive operations that will only be triggered under special circumstances such as at a specific location or in a certain time period. Additionally, HiSenDroid goes one step deeper to provide details aiming at helping security analysts understand why a given hidden sensitive operation is flagged as such. Experimental results over 20,000 apps, including both malicious and benign apps, show that hidden sensitive operations are indeed quite frequently presented in Android apps and HiSenDroid is effective in automatically discovering them. I further experimentally showed that, with FlowDroid, some of the hidden sensitive behaviours would eventually lead to private data leaks. Those leaks would have been hard to spot either manually among the large number of false positives reported by the state-of-the-art static analyzers, or by dynamic tools. Overall, by putting the light on hidden sensitive operations, HiSenDroid helps security analysts in validating potential sensitive data operations, which would be previously unnoticed.

- *This work has led to a research paper –* ***Demystifying Hidden Sensitive Operations in Android apps****, which has been published in the ACM Transactions on Software Engineering and Methodology (TOSEM) in 2022.*

JUNITTESTGEN (Chapter5): Despite being one of the largest and most popular projects, the official Android framework has only provided test cases for less than 30% of its APIs, leaving the majority of APIs uncovered. This poor test coverage of Android APIs has led to serious compatibility issues in the Android ecosystem, as recently shown [Li et al., 2018b, Wei et al., 2016, Mutchler et al., 2016, Zhang et al., 2015, Ham and Park, 2011, Huang et al., 2018].

To the best of our knowledge, the state-of-the-art works targeting compatibility issue detection leverage static analysis techniques to achieve their purpose. However, as known to the community, the static analysis will likely yield false positive results as it has to make some trade-offs when handling complicated cases (e.g., object-sensitive vs. object-insensitive). In addition, the static analysis will also likely suffer from soundness issues

because some complicated features (e.g., reflection, obfuscation, and hardening) are difficult to be handled [Sun et al., 2021b, Samhi et al., 2022]. Moreover, static app analysis can only be leveraged to perform post-momentum analysis (i.e., after the compatibility issues are introduced to the community). They cannot stop the problems from being distributed into the community – many Android apps, including very popular ones, still suffer from compatibility issues.

To mitigate this problem, I presented a prototype tool, JUnitTestGen, that attempts to automatically generate test cases for Android APIs based on their practical usage in real-world apps. Specifically, after locating existing API usages in real-world Android apps, JUnitTestGen performs field-aware, inter-procedural backward data-flow analysis to infer the API caller instance and its parameter values. JUnitTestGen then leverages the inferred information to reconstruct a minimal executable code snippet for the API under testing. Experimental results on thousands of Android apps show that JUnitTestGen is effective in generating test cases for Android APIs. It achieves an 80.4% of success rate in generating valid test cases. These test cases subsequently allow our approach to pinpoint various types of compatibility issues, outperforming a state-of-the-art generic test generation tool named EvoSuite, which can only generate test cases to reveal a small subset of compatibility issues. Furthermore, I demonstrated the usefulness of JUnitTestGen by comparing it against a state-of-the-art static analysis-based compatibility issue detector called CiD. JUnitTestGen is able to mitigate CiD's false-positive results and go beyond CiD's capability (i.e., detecting compatibility issues induced by APIs' signature changes) to detect compatibility issues induced by APIs' semantic changes.

- *This work has led to a research paper –* ***Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues****, which has been published at the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE) in 2022.*

1.4 Key Thesis Contributions

This Ph.D. thesis proposes three novel tools, i.e., SEEKER, HISENDROID, and JUNITTESTGEN to better detect specific security and compatibility issues in Android apps. These three tools deal with Android app security/compatibility issues from three perspectives: (i) helping conduct sensor-focused static taint analyses directly on the Android apps' bytecode and reports not only sensor-triggered privacy leaks but also the sensor types involved in the leaks; (ii) demystifying Hidden Sensitive Operations in Android apps, helping security analysts in validating potential sensitive data operations, which would be previously unnoticed; (iii) mining Android API usage to generate unit test cases for pinpointing compatibility issues.

Chapter3 introduces the **SEEKER**, a tool extends the famous FlowDroid tool to detect sensor-based data leaks in Android apps. I have made the following contributions:

1. I have designed and implemented a prototype tool, SEEKER (Sensor leak finder), that leverages static analysis to automatically detect privacy leaks originated from Android sensors. Although there are related works on sensor usage analysis, to the best of our knowledge, there is no other work that thoroughly analyses Android sensor leakage. Unlike previous works, our tool is the first one to characterize all kinds of sensor leaks in Android apps.

2. I applied SEEKER to analyze both malware and benign apps at a large scale. Our results show many sensor leaks that are overlooked by the state-of-the-art static analysis tool.

3. I have demonstrated the effectiveness of our tool by evaluating the sensor leaks it highlights.

4. I extended FlowDroid for supporting field sources detection (i.e., merged to Flow-

Droid via pull #385 on Github[Xiaoyu Sun, 2021]), which I believed could be adapted to analyze other sensitive field-triggered leaks. To benefit our fellow researchers and practitioners towards achieving this, I have made our approach open source at the following Github site.

Chapter4 introduces the **HISENDROID**, a tool for demystifying Hidden Sensitive Operations in Android apps. In this work, I have made the following contributions:

1. I proposed using a static analysis approach to discover hidden sensitive operations that are not exposed to the state-of-the-art static and dynamic analysis tools in Android apps. To this end, I leverage control flow and data flow analyses to identify the unique code level characteristics of hidden sensitive operations.

2. I designed and implemented a prototype tool HiSenDroid for analyzing hidden sensitive operations. I release HiSenDroid as an open source project [Sun, 2022] for supporting security analysts in their analysis needs and fostering further researches in this direction.

3. I evaluated HiSenDroid on a large-scale dataset that contains 10,000 benign and 10,000 malware samples, and discovered emerging anti-analysis techniques employed by malware samples, such as fulfilling certain restrictions related to *time*, *location*, *SMS message*, *system properties*, *package manager*, and other logics.

4. With the help of FlowDroid [Arzt et al., 2014], a static taint analyzer, I further experimentally showed that HSOs have been recurrently leveraged by attackers to leak sensitive user information.

Chapter5 introduces the **JUNITTESTGEN**, a tool for mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues. In this work, I have made the following contributions:

1. I have designed and implemented a prototype tool JUnitTestGen that leverages a novel approach to automatically generate unit test cases for APIs based on their existing usages.

2. I have set up a reusable testing framework for pinpointing API-induced compatibility issues by automatically executing a large set of unit test cases on multiple Android devices.

3. I have demonstrated the effectiveness of JUnitTestGen by i) generating valid test cases for Android APIs and pinpointing problematic APIs that could induce compatibility issues if accessed by Android apps, ii) outperforming state-of-the-art tools on real-world apps in detecting a wider range of compatibility issues.

With the help of our proposed tools, (i) developers can use our tool SEEKER to automatically detect sensor leaks; our fellow researchers and practitioners can also use our tool to analyze other sensitive field-triggered leaks; (ii) by putting the light on hidden sensitive operations, security analysts can use our tool HISENDROID in validating potential sensitive data operations, which would be previously unnoticed; (iii) Android app developers and google developers can use our tool JUNITTESTGEN to detect zero-day compatibility issues, including not only signature-based but also semantics-based compatibility issues.

1.5 Structure of the thesis

In this thesis, Chapter 1 introduces the introduction and motivation about the thesis. Chapter 2 surveys the related literature which includes specific security and compatibility issues in Android apps, such as sensor leaks, hidden sensitive operations, signature-based and semantics-based compatibility issues. In Chapter 3, I present **SEEKER**, an approach to automatically detect sensor leaks in Android apps. In Chapter 4, I present **HISENDROID**, an approach to demystify Hidden Sensitive Operations in Android apps. In Chapter 5, I

present **JUNITTESTGEN**, a tool that mines Android API usage to generate unit test cases for pinpointing compatibility issues. Chapter 6 gives the conclusion, some limitations and future research directions.

Chapter 2

Literature Review

In this chapter, I provide the necessary background to understand the purpose, key concerns, and technical details of the three research studies I conduct in this PhD research. I conduct a systematic literature review to identify state-of-the-art works in detecting security and compatibility issues in Android applications. I categorize the relevant research into three categories: static analysis of sensor leak detection on Android apps, evasive techniques detection, and compatibility issues detection in Android apps. It can help the readers to get a systematic overview of the work in general and specifically to know future research directions.

2.1 Static analysis for sensor leak detection

Android sensor usage: Android sensor usage has long been analyzed in software security research. Several related works [Zhu et al., 2013, Ba et al., 2020, Xu et al., 2012, Miluzzo et al., 2012, Liu et al., 2015a, Aviv et al., 2012, Cai and Chen, 2011, Owusu et al., 2012, Lee and Lee, 2015] have indicated that embedded sensors can be intentionally misused by malicious apps to compromise privacy. Ba et al. [Ba et al., 2020] proposed a side-channel attack that adopts accelerometer data to eavesdrop on the speaker in smartphones. Xu et al. [Xu et al., 2012] have shown that it is feasible to infer user's tap inputs using its integrated motion sensors. Liang Cai et al.[Cai and Chen, 2011] revealed that confidential data could be leaked when motion sensors, such as accelerometers and gyroscopes, are used to infer keystrokes. Also, Lin et al.[Lin et al., 2012] demonstrated that the orientation sensor of the

smartphone could be utilized to detect users' unique gestures used to hold and operate their smartphones.

Android Sensor misuse is one of the major causes of privacy leaks and security issues on the Android platform. Zhu et al. [Zhu et al., 2013] collected sensor data from accelerometers, gyroscopes and magnetometers and constructs users' gesture based on these data. Their work indicates that it is feasible to get access to sensory data for personal identification and behaviour analysis. Liu et al. [Liu et al., 2015a] demonstrated the most frequently used sensors in Android devices and revealed their usage patterns through backwards tracking analysis. They further investigated sensor data propagation path for accurately characterizing the sensor usage.

Software side-channels attacks: Many previous studies [Chang et al., 2009, Lester et al., 2004, Liu et al., 2009, Ravi et al., 2005, Allen et al., 2006, Schlegel et al., 2011] explored password inference through specific sensors on smartphones. Owusu et al. [Owusu et al., 2012] showed that accelerometer values could be used as a powerful side channel attack to figure out the password for a touchscreen keyboard. Cai et al.[Cai and Chen, 2011] provided insights into how motion sensors, such as accelerometers and gyroscopes, can be used to infer keystrokes. Cai et al. [Cai et al., 2009] found that mobile phone sensors are inadequately protected by the Android permission system and thus can raise serious privacy concerns. Enck et al. [Enck et al., 2014] developed TaintDroid that takes sensor information (i.e., location and accelerometer) as sources to detect privacy leaks. Mehrnezhad et al. [Mehrnezhad et al., 2018] show that orientation sensor can be stealthily listened to without requesting any permission, contributing for attackers to infer the user's PIN. However, these works only present the challenges facing in the detection of sensor-sniffing apps or only provided specific attacks by using sensor data. None of them can systematically characterize data leaks in all kinds of sensors.

Static analysis of Android apps for security: Android users have suffered from

privacy leaks [Li et al., 2017a, Kong et al., 2018, Samhi et al., 2021, Octeau et al., 2016]. Several solutions have been proposed for detecting such data leaks through static taint analysis [Gao et al., 2020, Li et al., 2015b, Yang et al., 2017]. For example, Arzt et al. [Arzt et al., 2014] developed FlowDroid, a context, flow, field and object-sensitive static analysis tool for detecting potential data leaks in Android Apps. Based on Soot [Vallée-Rai et al., 2010], FlowDroid relies on pre-defined knowledge to pinpoint taint flows between source and sink APIs. Zhang et al. [Zhang et al.,] developed ConDySTA, a dynamic taint analysis approach, as a supplement to static taint analysis by introducing inaccessible code and sources that help reduce false negatives. Further, Li et al. [Li et al., 2015c] presented IccTA, which can precisely perform data-flow analysis across multiple components in Android apps. Klieber et al. [Klieber et al., 2014] augment the FlowDroid and Epicc[Octeau et al., 2013] analyses by tracking both inter-component and intra-component data flow in Android apps. However, none of these tools focuses on the leaks that originate from sensors.

The most similar work to our proposed SEEKER tool is SDFDroid[Liu et al., 2018], which provides sensor usage patterns through data flow analysis. As a static analysis approach, however, it focuses on different research objectives compared to SEEKER. For example, SDFDroid reveals sensor usage patterns while our work explores how and where the sensor data are leaked. On the other hand, SDFDroid applies a static approach to extract sensor data propagation path to construct sensor usage patterns through clustering analysis. In contrast to SDFDroid, SEEKER provides detailed privacy leaks caused by misuse of sensor data, which haven't been found by SDFDroid.

2.2 Evasive techniques detection

Hidden sensitive operations have long existed in Android malware as evasive technologies have widely been used by attackers to hide malicious behaviors. The research community has hence proposed various approaches to tackle these issues. I now discuss some of the

representative works from two angles: the evasive techniques that have been proposed to hide malicious code from being identified, and the detection methods proposed to pinpoint such evasive techniques.

Evasive Techniques: There have been a number of research works conducted on hiding malicious behavior from detection, most of which focus on evading the dynamic test platforms such as virtual machines and emulators. Early works target the Windows platform [Chen et al., 2008], while recently the trend has been moved to Android [Petsas et al., 2014, Vidas and Christin, 2014, Jing et al., 2014, Diao et al., 2016, Costamagna et al., 2018, Liu et al., 2021c, Liu et al., 2022c]. These evasive techniques detect the presence of a simulated environment by either looking into the system properties of the testing platform (e.g., system fingerprints, hardware capabilities, etc.) [Petsas et al., 2014, Vidas and Christin, 2014, Costamagna et al., 2018], or leveraging a reverse Turing test that examines if the app interacts with a human user [Diao et al., 2016]. For instance, Diao et al. [Diao et al., 2016] observed that programmed interaction has specific patterns of input and interaction frequency, which is different from real users. Overall, the evasive techniques usually hide malicious activities in an *if-then-else statement*. The hidden malicious behavior will only be set off when certain conditions are fulfilled (e.g., not in an emulator); otherwise dummy benign operations are triggered. The prevalence of such evasive techniques motivated us to investigate the HSOs in Android apps and propose HiSenDroid to detect them.

Detection of Evasive Techniques: The pervasive evasive techniques (e.g., anti-emulator techniques) have motivated the research community to take countermeasures. Great effort has been spent on detecting known types of hidden behaviors that hampers the dynamic analysis process. These works include detecting anti-emulator techniques [Balzarotti et al., 2010, Lindorfer et al., 2011, Kirat et al., 2014, Kirat and Vigna, 2015, Ami et al., 2022] and generic logic-bombs [Crandall et al., 2006, Brumley et al., 2008, Zheng et al., 2012, Fratantonio et al., 2016, Papp et al., 2019]. The approaches used for detecting

anti-emulator techniques compare the behavioral deviation of the tested apps on the various environments when feeding them the same input. The fundamental idea is that if the app behaves differently in different environments, it is likely trying to evade one or more analysis platforms (usually referred to as bare-metal analysis in the literature) [Balzarotti et al., 2010, Lindorfer et al., 2011, Kirat et al., 2014, Kirat and Vigna, 2015]. While these early works investigate a critical category of hidden operations (i.e., anti-emulator), their proposed methods lack generalization that cannot be applied to detect other types of hidden operations emerging recently. In addition, many flaws have been found in traditional static analysis tools of evasive techniques detection [Ami et al., 2021, Bonett et al., 2018], including missing callbacks, missing implicit calls, incorrect modeling of anonymous classes and incorrect modeling of asynchronous methods, etc.

Besides the detection of anti-emulator techniques, several works focused on uncovering other trigger-based behaviors. These approaches leverage symbolic execution or static code analysis and instrumentation to expose the hidden branches in an *if-then-else statement* [Crandall et al., 2006, Brumley et al., 2008, Zheng et al., 2012, Fratantonio et al., 2016, Papp et al., 2019]. As examples, Zheng et al. [Zheng et al., 2012] proposed to leverage a static analysis approach to retrieve all UI related events, and use dynamic testing to trigger them and log the invocation of sensitive APIs. Unlike HiSenDroid that leverages static analysis, the dynamic analysis based approach introduces significant system- and time-overhead. The coverage of the dynamic analysis is also in question. Fratantonio et al. [Fratantonio et al., 2016] proposed TriggerScope to detect hidden triggered behaviors based on the observation that certain triggers (i.e., time, location, and SMS related triggers) always involve the comparison of specific types of input (i.e., system time, system location, and received SMS). Symbolic execution is then leveraged to detect such narrow conditions. While TriggerScope is effective in detecting the above-mentioned three types of logic bombs, it cannot be generalized to detect hidden operations triggered by other

types of conditions, such as system property, which has been found pervasive in Android apps.

Similar to our proposed HiSenDroid, another line of work attempts to detect unknown types of trigger-based behaviors [Pan et al., 2017], [Wang et al., 2017]. A prominent example is HSOMiner [Pan et al., 2017], which extracts static characteristics of hidden behaviors as features and trains a machine learning model to identify the code blocks that observe similar patterns. The major differences between our work and HSOMiner are twofold. First, HSOMiner requires a large number of manually labelled training samples, which involves extensive human experts' effort. Its performance also heavily relies on the manually labelled training data, which is prone to errors. Our method, on the other hand, is an automatic process without human intervention. Second, HSOMiner, as a machine learning based approach, lacks explanations of the decisions. In contrast, our static code analysis based approach outputs the full call traces of detected HSOs, and provides more detailed information for further analysis.

2.3 Compatibility issues detection

Android API evolution is a critical issue in software maintenance [Nagappan and Shihab, 2016, Li et al., 2018a, Martin et al., 2016, Li et al., 2016a, Oliveira et al., 2018, Lamothe et al., 2020, Li et al., 2018c, Zhou and Walker, 2016, Dig and Johnson, 2005, Kapur et al., 2010, Liu et al., 2022b, Sun et al., 2021a]. McDonnell et al. [McDonnell et al., 2013] have shown that the Android system updates 115 APIs per month on average, while app developers usually adopt the new APIs at a much more slower rate. The slow adoption of API updates may raise various issues, such as security and compatibility. An empirical study on StackOverflow conducted by Linares-Vasquez et al. [Linares-Vásquez et al., 2014] suggests that API updates would trigger more discussions, especially if APIs are removed from the Android system. They also revealed that users are in more favour of

apps that use less fault and change-prone APIs [Linares-Vásquez et al., 2013, Bavota et al., 2014], as these apps would likely introduce fewer failures, crashes and other bugs.

Android Fragmentation is one of the major causes of compatibility and security issues on the Android platform. Han et al. [Han et al., 2012] analyzed the bug reports on two Android smartphone vendors, HTC and Motorola. Wu et al. [Wu et al., 2013] studied ten customized Android images across five vendors. Their findings suggest that system customization brings serious security risks through over-privileged system apps and introduced vulnerabilities. Zhou et al. [Zhou et al., 2014] further investigated the security issues in customized Android systems, and proposed a tool to detect such issues. Liu et al. [Liu et al., 2014] studied the characteristics of bugs in Android apps and observed that certain Android performance bugs are tied to specific devices.

Android developers have long been suffered from compatibility issues due to the fast-evolving and fragmented nature of the Android ecosystem [Xia et al., 2020, Kamran et al., 2016, Li et al., 2017a, Nayebi et al., 2012, Zhao et al., 2022]. Researchers have proposed several solutions for detecting compatibility issues of Android APIs. Wei et al. [Wei et al., 2016, Wei et al., 2018] conducted an empirical study to investigate fragmentation-induced API compatibility issues and proposed a tool named FicFinder to detect such APIs. FicFinder identifies APIs with compatibility issues based on heuristic rules manually derived from a limited number of Android apps, which is expected to introduce high false negatives (i.e., missing undiscovered compatibility issues). Subsequently, several new works have been proposed to leverage data-driven techniques that automatically mine compatibility issues from various sources such as Android code base and real-world apps. Li et al. proposed a tool named CiD to detect potential compatibility issues by mining the history of the Android framework source code. CiD identifies Android APIs' lifetimes and finds if an app's declared supported versions conflict with its used APIs. He et al. [He et al., 2018] proposed IctApiFinder that utilizes a similar strategy to extract an API's lifetime usage and

introduce inter-procedural data flow analysis to locate incompatible APIs, which reduces the false-positive results produced by CiD.

Comparable to our method, several works have been proposed to mine API usage from real-world apps. Scalabrino et al. [Scalabrino et al., 2019, Scalabrino et al., 2020] considered the APIs wrapped in a version check condition (e.g., *if (Build.VERSION.SDK_INT >= 21)*) to potentially have compatibility issues and developed a tool named ACRYL to extract such APIs from real-world apps. However, ACRYL can only detect APIs whose compatibility issue is already known by the developers (i.e., they are enclosed in the version check conditions by the developers), while our method is capable of detecting zero-day compatibility issues that the app developers are not yet aware of, or even Google itself. Other generic test generation tools, such as EvoSuite and Randoop[Pacheco and Ernst, 2007], are able to generate tests for Java classes. However, these tools do not directly aim at generating tests for APIs, and they have been demonstrated as insufficient in pinpointing compatibility issues because of the lack of API usage knowledge.

Another work proposed by Wei et al. [Wei et al., 2019] shares a similar idea to detect compatibility issues that may arise on specific devices introduced by the customization of the Android system by OEMs. They proposed a tool named Pivot that finds APIs enclosed in device-checking statements (e.g., *if (android.os.Build.Model.equals("Nexus 6P"))*) from popular Android apps, and construct API-device correlations to imply that the API may have compatibility issues on the given device. Our method, though not specifically designed for detecting device-related compatibility issues, is also able to detect such issues. Moreover, human effort is involved in validating the API-device correlation identified by Pivot, which is prone to error. Our method, on the other hand, is fully automated without human intervention while having a true-positive rate of 100%.

Chapter 3

Characterizing Sensor Leaks in Android Apps

Xiaoyu Sun, Xiao Chen, Kui Liu, Sheng Wen, Li Li, and John Grundy. 2021. Characterizing Sensor Leaks in Android Apps. The IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE) 2021. https://arxiv.org/pdf/2201.06235.pdf

3.1 Introduction

As of 1st January 2021, there are nearly three million Android apps available on the official Google Play app store. The majority of them (over 95%) are made freely accessible to Android users and cover every aspect of users' daily life, such as supporting social networking, online shopping, banking, etc. Many of these functionalities are supported by application interfaces provided by the Android framework, essentially fulfilled by a set of hardware-based sensors [Google, 2021b]. For example, Android apps often leverage accelerometer sensors to detect the orientation of a given smartphone and user movement, and the temperature sensor to detect the device's temperature.

Despite being needed to support the implementation of many diverse Android apps, mobile phone sensors can also be abused to achieve malicious behaviors. There have been many reports of apps that exploit sensors in Android devices to conduct malicious activities. For example, Adam et al. [Aviv et al., 2012] have experimentally shown that the accelerometer sensor could be leveraged as a side-channel to infer mobile users' tap and gesture-based input. Xu et al. [Xu et al., 2012] have also demonstrated the possibility of this attack by presenting to the community a Trojan application named *TapLogger* to silently infer user's tap inputs based on the device's embedded motion sensors. Similarly, Schlegel

et al. [Schlegel et al., 2011] have provided another Trojan application called *Soundcomber* that leverages the smartphone's audio sensor to steal users' private information.

These studies have experimentally shown that the leaks of Android sensor data can cause severe app security issues. We argue that there is thus a strong need to invent automated approaches to detect such sensor leaks in Android apps before publishing them onto app markets. To the best of our knowledge, existing works focus on detecting certain types of sensor usage and its corresponding suspicious behaviors. None of them are designed as a generic approach for systematically revealing data leaks in all types of Android sensors. Also, these works mainly concentrate on discovering and understanding the usage patterns of Android embedded sensors, which do not involve completed data flow analysis to pinpoint sensitive data leaks caused by sensors.

Although many generic approaches to detect privacy leaks in Android apps have been proposed, none can be directly applied to achieve our purpose, i.e., detecting generic sensor leaks in Android apps. Indeed, the famous FlowDroid tool has been demonstrated to be effective in detecting method-based privacy leaks in Android apps. It performs static taint analysis on Android apps' bytecode and attempts to locate data-flow paths connecting two methods, i.e., from a *source* to a *sink* method. Here, *source* refers to such methods that obtain and return sensitive information from the Android framework (e.g., get device id), while *sink* refers to such methods that perform dangerous operations such as sending data to remote servers. FlowDroid has been designed as a generic approach. It has provided a means for users to pre-define the needed *source* and *sink* methods. Unfortunately,FlowDroid does not allow users to configure fields as *sources* so as to support the detection of privacy leaks flowing from *fields* to sensitive operations (i.e., *sink*). Since sensor data in Android is mostly provided via fields, FlowDroid cannot be directly applied to detect sensor leaks in Android apps.

To address this research gap, we designed and implemented a prototype tool, SEEKER,

to automatically detect sensor data leaks in Android apps. We extend the open-source tool FlowDroid to support field-triggered sensitive data-flow analyses. Our new SEEKER further performs a detailed static code analysis to infer the sensor types involved in the sensitive data-flows as the leaked sensor data is not directly associated with the sensor type. (we detail this challenge in Section 3.4.3). We then apply SEEKER to detect and characterize sensor leaks in real-world Android apps. Based on 40,000 randomly selected Android apps, including 20,000 benign apps and 20,000 malicious apps, our experimental results show that SEEKER is effective in detecting sensor leaks in Android apps. We also find that malware is more interested in obtaining and leaking sensor data than benign apps, and Accelerometer and Magnetic are among the most targeted sensors by those malicious apps.

In summary, this work makes the following three main contributions:

- We have designed and implemented a prototype tool, SEEKER (Sensor leak finder), that leverages static analysis to automatically detect privacy leaks originated from Android sensors.

- We apply SEEKER to analyze both malware and benign apps at a large scale. Our results show many sensor leaks that are overlooked by the state-of-the-art static analysis tool.

- We have demonstrated the effectiveness of our tool by evaluating the sensor leaks it highlights.

The rest of this chapter is organized as follows.

Section 3.2 presents key related work on question generation and relevant techniques. Section 3.3 presents the motivation of this study. Section 3.4 presents the details of our approach for the question generation task in Stack Overflow.

Section 3.5 presents the experimental setup, the baseline methods and the evaluation metrics used in our study. Section 3.6 presents the detailed research questions and the

evaluation results under each research question. Section 3.7 presents the contribution of this study and discusses the strength and weakness of this study. Section 3.8 presents threats to validity of our approach. Section 3.9 concludes this study with possible future work.

3.2 Related Work

3.2.1 Android sensor usage

Android sensor usage has long been analyzed in software security mechanisms. Related works [Zhu et al., 2013, Ba et al., 2020, Xu et al., 2012, Miluzzo et al., 2012, Liu et al., 2015a, Aviv et al., 2012, Cai and Chen, 2011, Owusu et al., 2012, Lee and Lee, 2015] have indicated that embedded sensors can be intentionally misused by malicious apps for privacy compromise. Ba et al. [Ba et al., 2020] proposed a side-channel attack that adopts accelerometer data to eavesdrop on the speaker in smartphones. Xu et al. [Xu et al., 2012] have shown that it is feasible to infer user's tap inputs using its integrated motion sensors. Liang Cai et al.[Cai and Chen, 2011] revealed that confidential data could be leaked when motion sensors, such as accelerometers and gyroscopes, are used to infer keystrokes. Also, Lin et al.[Lin et al., 2012] demonstrated that the orientation sensor of the smartphone could be utilized to detect users' unique gesture to hold and operate their smartphones.

Android Sensor misuse is one of the major causes of privacy leaks and security issues on the Android platform. Zhu et al. [Zhu et al., 2013] collected sensor data from accelerometers, gyroscopes and magnetometers and constructs users' gesture based on these data. Their work indicates that it is feasible to get access to sensory data for personalized usage. Liu et al. [Liu et al., 2015a] demonstrated the most frequently used sensors in Android devices and revealed their usage patterns through backward tracking analysis. They further investigate sensor data propagation path for accurately characterizing the sensor usage [Liu et al., 2018]. Their findings suggest that the accelerometer is the most frequently

used sensor and the sensor data are always used in local codes.

3.2.2 Software side-channels attacks.

Many previous studies [Chang et al., 2009, Lester et al., 2004, Liu et al., 2009, Ravi et al., 2005, Allen et al., 2006, Schlegel et al., 2011] explored password inference through specific sensors on smartphones. Owusu et al. [Owusu et al., 2012] showed that accelerometer values could be used as a powerful side channel to figure out the password on a touchscreen keyboard. Cai et al.[Cai and Chen, 2011] provided insights of how motion sensors, such as accelerometers and gyroscopes, can be used to infer keystrokes. Cai et al. [Cai et al., 2009] found that mobile phone sensors are inadequately protected by permission system so that it can raise serious privacy concerns. Enck et al. [Enck et al., 2014] developed TaintDroid that takes sensor information (i.e., location and accelerometer) as sources to detect privacy leaks. Mehrnezhad et al. [Mehrnezhad et al., 2018] show that orientation sensor can be stealthily listened to without requesting any permission, contributing for attackers to infer the user's PIN. However, these works emphasize the challenges facing the detection of sensor-sniffing apps or only provided specific attacks by using sensor data. None of them can systematically characterize data leaks in all kinds of sensors.

3.2.3 Static analysis on Android apps.

Android users have long been suffered from privacy leaks [Li et al., 2017a, Kong et al., 2018, Samhi et al., 2021, Octeau et al., 2016]. Several solutions have been proposed for detecting such data leaks through static taint analysis [Gao et al., 2020, Li et al., 2015b, Yang et al., 2017]. For example, Arzt et al. [Arzt et al., 2014] developed FlowDroid, a context, flow, field and object-sensitive static analysis tool for detecting potential data leaks in Android Apps. Based on Soot [Vallée-Rai et al., 2010], FlowDroid relies on pre-defined knowledge to pinpoint taint flows between source and sink APIs. Zhang et al. [Zhang et al.,] developed ConDySTA, a dynamic taint analysis approach, as a supplement

to static taint analysis by introducing inaccessible code and sources that help reduce false negatives. Further, Li et al. [Li et al., 2015c] presented IccTA, which can precisely perform data-flow analysis across multiple components for Android apps. Klieber et al. [Klieber et al., 2014] augment the FlowDroid and Epicc[Octeau et al., 2013] analyses by tracking both inter-component and intra-component data flow in Android apps. However, none of these tools concerns the leaks that originated from sensors. Apart from that, our tool only takes the sensor-related code into account, which cost less time by pruning the control flow graph.

The most similar work to ours is SDFDroid[Liu et al., 2018], which provides the sensor usage patterns through data flow analysis. As a static approach, however, it focuses on different research object compared to SEEKER. For example, SDFDroid reveals sensor usage patterns while our work explores how and where the sensor data are leaked. On the other hand, SDFDroid applies a static approach to extract sensor data propagation path to construct sensor usage patterns through clustering analysis. In contrast to SDFDroid, SEEKER provide detailed privacy leaks caused by misuse of sensor data, which haven't been found by SDFDroid.

3.3 Motivation

Sensors have been widely adopted for launching side-channel attacks against smart devices [Sikder et al., 2021]. Table 3.1 summarizes a diverse set of sensor-based attacks targeting smartphones and smartwatches. Since accessing sensitive sensor data does not require any security checks (e.g., permission check), attackers can easily trigger malicious behaviors by making use of such data. As revealed in the table, generally, sensor leakage are performed with the aim of (1) keystroke inference, (2) task inference (refers to a type of attack which reveals the information of an on-going task or an application in a smart device), (3) location inference, and (4) eavesdropping. For example, motion and position sensors can

be exploited for keystroke inference, leading to severe privacy leaks such as passwords, credit card information, etc. Light sensor is found to eavesdrop acoustic signals in the vicinity of the device, causing private information leak. Magnetic sensors can be exploited to compromise electromagnetic emanations, which would affect the confidentiality of the devices.

Table 3.1: Examples of Sensor-based Cybersecurity attacks.

Sensor Category	Sensor Type	Attack Description
Motion sensor	Accelerometer	sniffing smartwatch passwords [Lu et al., 2018]
	Accelerometer, Gyroscope	Text Inference [Hodges and Buckley, 2018]
	Accelerometer, Gyroscope	Motion-based keystroke inference [Cai and Chen, 2012]
	Accelerometer, Gyroscope	Keystroke inference on Android [Al-Haiqi et al., 2013]
	Accelerometer	Accelerometer side channel attack [Aviv et al., 2012]
	Accelerometer	Touchscreen area identification [Owusu et al., 2012]
	Accelerometer	Decoding vibrations from nearby keyboards [Marquardt et al., 2011]
	Gyroscope	Single-stroke language-agnostic keylogging [Narain et al., 2014]
	Accelerometer, Gyroscope	Inferring Keystrokes on Touch Screen [Cai and Chen, 2011]
	Accelerometer, Gyroscope	Inferring user inputs on smartphone touchscreens [Xu et al., 2012]
	Accelerometer, Gyroscope	Keystroke Inference [Bo et al., 2019]
	Accelerometer	keystrokes Inference in a virtual environment. [Ling et al., 2019]
	Accelerometer, Gyroscope	Risk Assessment of motion sensor [Huang et al., 2019]
	Accelerometer, Gyroscope	Infer tapped and traced user input [Nguyen, 2015]
	Accelerometer, Gyroscope	Motion-based side-channel attack [Lin and Seibel, 2019]
	Accelerometer	Keystroke inference with smartwatch [Liu et al., 2015b]
	Accelerometer	Motion leaks through smartwatch sensors [Wang et al., 2015]
	Accelerometer	Side-channel inference attacks [Maiti et al., 2018b] [Maiti et al., 2015]
	Accelerometer	Smartphone PINs prediction [Sarkisyan et al., 2015]
	Gyroscope	Inferring Mechanical Lock Combinations [Maiti et al., 2018a]
	Accelerometer, Gyroscope	Inference of private information [Maiti et al., 2018a]
	Accelerometer, Gyroscope	Typing privacy leaks via side-Channel from smart watch [Liu and Li, 2019]
	Accelerometer, Magnetometer	Input extraction via motion sensor [Shen et al., 2015]
	Gyroscope	Recognizing speech [Michalevsky et al., 2014]
Position sensor	Magnetic	Compromising electromagnetic emanations [Vuagnoux and Pasini, 2009]
	Magnetic	My Smartphone Knows What You Print [Song et al., 2016]
	Magnetic	Location detection [Block and Noubir, 2018]
Environment sensor	Light Sensor	Optical eavesdropping on displays [Chakraborty et al., 2017]

As a concrete example, Lu et al. [Lu et al., 2018] revealed that sensitive intercepting password could be accessed through motion data on the smartwatch's onboard sensors. They proposed *Snoopy*, a password extraction and inference approach via sensor data for PIN attack, which could affect smartwatch users in a non-invasive way. *Snoopy* extracts the segments of motion data when users entered passwords and then applies deep learning techniques to infer the actual passwords. Figure 1.1 gives two examples of the differences of the motion sensor data changes when the user swipes or taps a password on a smartwatch. *Snoopy* demonstrates the feasibility of sensor data leaks by intercepting password

information entered on smartwatches. Such real-world sensor-enabled attacks motivated us to provide automatic tools for characterizing universal sensor leaks in Android Apps that have been long overlooked.

3.4 Approach

This work aims to automatically detect information leaks of onboard sensors in Android apps. To this end, we design and implement a prototype tool called SEEKER for achieving this purpose. Figure 3.1 describes the overall working process of SEEKER, which is mainly made up of three modules, namely Sensitive Sensor Source Identification, Sensor-triggered Static Taint Analysis and Sensor Type Inference.

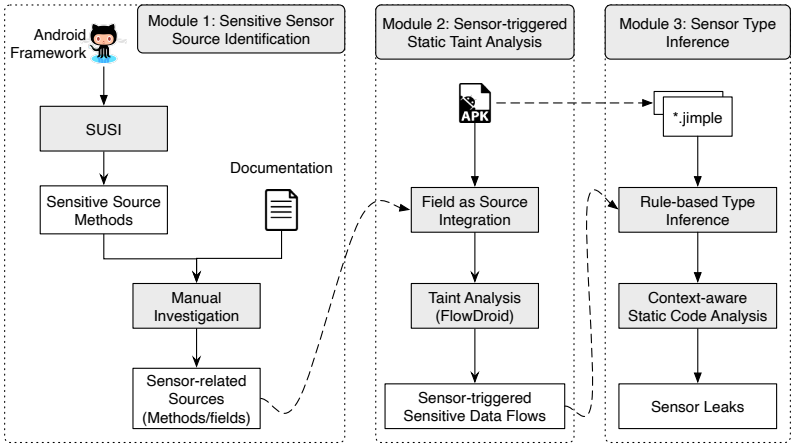


Figure 3.1: The working process of our approach.

3.4.1 Sensitive Sensor Source Identification

The first module, *Sensitive Sensor Source Identification*, aims to identify sensor-related sources that access and obtain sensitive information related to the device's sensors. As reported by Liu et al. [Liu et al., 2018], Android sensor data can be obtained through invoking sensor-related APIs or directly accessing local fields in which the sensor data is stored. In this work, we take both of these types into consideration, aiming at pinpointing all the possible sensor-triggered privacy leaks.

To do this we need to identify all the sensor-related sources, including both Android methods and fields. For Android methods, we use the well known SUSI tool [Arzt et al., 2013] to obtain sensor-related source methods. SUSI is a novel machine-learning guided approach that scans Android API's source code to predict *source* and *sink* methods, based on a training set of hand-annotated sources and sinks. In this work, we launch SUSI on the latest Android Open Source Project (i.e., AOSP version 11.0) and manually filter out non-sensor related source methods.

To identify sensor-related fields (as sources), there is no existing approach to achieve such a purpose. We resort to a manual process of going through the Android Developers' Documentation to identify source fields storing sensitive sensor information. The identified fields are then discussed and confirmed by the authors by measuring whether leaking such information would potentially expand the attack surface to users' privacy. Finally, we identified 79 fields and 20 methods as the sources. Table 3.2 lists the selected sources that indeed introduce leaks in our experimental dataset. A full list of field and method sources can be found in the *SourcesAndSinks.txt* file of our open-source project[1].

Table 3.2: The list of sensitive sensor sources.

Sensor-related Source	Source Type
SensorEvent#values	Field
SensorEvent#timestamp	Field
Sensor#getName()	Method
Sensor#getVendor()	Method
Sensor#getVersion()	Method
SensorManager#getDefaultSensor(int)	Method
Sensor#getMaximumRange()	Method
SensorManager#getSensorList(int)	Method
Sensor#getType()	Method
Sensor#getResolution()	Method
Sensor#getPower()	Method

[1]https://github.com/MobileSE/SEEKER

3.4.2 Sensor-triggered Static Taint Analysis

The ultimate goal of SEEKER is to detect sensor-related data leaks. To this end, we implement the *Sensor-triggered Static Taint Analysis* module that extends state-of-the-art tool FlowDroid [Arzt et al., 2014] to facilitate sensor-related data leak detection. FlowDroid detects data leaks by computing data flows between sources and sinks. FlowDroid defined a sensitive data flow happens when a suspicious "tainted" information passes from a source API (e.g., `getDeviceId`) to a sink API (e.g., `sendTextMessage`).

FlowDroid is a state-of-the-art tool and it provides a highly precise static taint-analysis model, especially for Android applications. However, FlowDroid only takes API statements as sources or sinks, leading to false negatives because of the lack of field-triggered sources. Thus, in this work, we extend FlowDroid by supporting field statement as sources, so as to pinpoint data leaks originated from specific field sources of interest.

Our preliminary study discovered that certain sensor-related data leaks are sourced from data stored in class fields (e.g., android.hardware.SensorEvent#values). We therefore implemented our own class that implements the `ISourceSinkDefinitionProvider` interface in FlowDroid for supporting the declaration of fields as sources. Also, based on the feature of class fields, we defined a new model names AndroidField extends from `SootFieldAndMethod`. After loading a specific field statement from *source&sink.txt* file, we apply a field pattern regular expression to convert it to the AndroidField model.

FlowDroid has the ability to compute data flow connections between all possible statements. In the implementation of FlowDroid, `ISourceSinkManager` interface marks all statements as possible sources and then records all taint abstractions that are passed into `getSourceInfo()`. To that end, we pass the constructed field model as a source statement to the following taint analysis process. In this way, sensitive data flow can be detected starting at given field source statements.

3.4.3 Sensor Type Inference

The primary goal of SEEKER is to detect data leaks from Android platform sensors. With the help of FlowDroid's taint analysis, SEEKER's second module can detect sensor-triggered sensitive data flows. Unfortunately for the field-triggered ones, the identified data-flows only show that there is sensor data leaked but do not tell from which sensor the data is collected. The sensor type information is important for helping security analysts understand the sensor leaks. Therefore, in our last module, we identify the types of sensors that are leaking information.

To identify which sensors exist on a specific Android device, we first get a reference to the sensor service by creating an instance of the `SensorManager` class via calling the `getSystemService()` method with *SENSOR_SERVICE* argument. After that, we can determine available sensors on the device by calling the `getSensorList()` method. The `getSensorList()` method returns a list of all available sensors on the device by specifying constant *TYPE_ALL* as the parameter. A list of all sensors from a given type can also be retrieved by replacing the parameter as the constants defined for corresponding sensor types, such as *TYPE_GYROSCOPE, TYPE_LINEAR_ACCELERATION*, etc. We can also determine whether a specific type of sensor exists by calling the `getDefaultSensor()` method with the target type constant (the same as the ones passed in to `getSensorList()` method). If a device has that type of sensor, it will return an object of that sensor. Otherwise, null will be returned.

We use a rule-based strategy to identify the sensor type of a leak in the case of only one sensor registered in the given app. To do this, SEEKER obtains the sensor type by looking into the type constant in the `getDefaultSensor()` statement. For instance, `getDefaultSensor(Sensor.TYPE _ACCELEROMETER)` indicates that the Accelerometer sensor is obtained. We can then reasonably assume that all sensor-related data leaks in the class are associated with the identified sensor (because only this sensor is

registered).

```
1 public class MainActivity extends AppCompatActivity implements SensorEventListener{
2 @Override
3 public void onSensorChanged(SensorEvent sensorEvent) {
4  switch(sensorEvent.sensor.getType()) {
5   case Sensor.TYPE_ACCELEROMETER:
6    accX = sensorEvent.values[0];
7    accY = sensorEvent.values[1];
8    accZ = sensorEvent.values[2];
9    ...
10  case Sensor.TYPE_GYROSCOPE:
11   gyroX = sensorEvent.values[0] * 5;
12   gyroY = sensorEvent.values[1] * 5;
13   gyroZ = sensorEvent.values[2] * 5;
14   ...
15  case Sensor.TYPE_ROTATION_VECTOR:
16   rvX = sensorEvent.values[0];
17   rvY = sensorEvent.values[1];
18   rvZ = sensorEvent.values[2];
19   ...
20 }}}
```

Listing 3.1: An example of sensor type usage with switch branch.

In the case of multiple sensors registered in the given app, we further leverage context-aware static code analysis to find the connection between sensor types and the leaked field data. Firstly, we locate the invocation statement of API `android.hardware.SensorManager #getDefault Sensor(int)` in the `onSensorChanged()` method. In the multiple sensors scenario, different sensor's behavior is handled in a conditional branch (e.g. if-then-else statement or switch statement). We then apply context-aware static code analysis to detect the code branch that contains the taint sensor source statement, based on which we then resolve the sensor type in the branch condition.

We further elaborate on the context-aware static code analysis with an example presented in Listing 3.1. The code snippet in the Listing shows an example of how multiple sensors are handled with `onSensorChanged(android .hardware.SensorEvent)` method. Android determines the activated sensor by matching the `sensorEvent.sensor.getType()` method (line 4). For example, if `get Type()` returns `Sensor.TYPE_ACCELEROMETER` (line 5), the data obtained by `sensorEvent.values` is associated with the Accelerometer sensor (lines 6-8); if `getType()` returns `Sensor.TYPE_GYROSCOPE` (line 10), the data contained in `sensorEvent.values` is accord-

ingly associated with the current activated sensor, i.e., Gyroscope (lines 11-13).

3.5 Experimental Setup

SEEKER is designed to expose the data leak issues of sensors in Android apps. We investigate the feasibility and effectiveness of detecting sensor leaks in Android apps with the following three research questions:

- **RQ1:** *Can SEEKER effectively detect sensor leaks in Android apps?* This research question aims to investigate the feasibility of detecting sensor leaks in Android apps with SEEKER.
- **RQ2:** *To what extent diverse sensor leaks can be identified by SEEKER?* With this research question, we explore the sensor types related to the identified sensitive data leaks, and investigate to what extent such sensor leaks are targeted by attackers.
- **RQ3:** *Is SEEKER efficient to detect the sensor leaks in Android apps?* In this study, we leverage the time costs of detecting sensor leaks to assess the efficiency of SEEKER.

To answer the aforementioned research questions, we build the experimental dataset with a *malware* set and a *benign* set. The *malware* set contains 20,000 Android apps including malware downloaded from VirusShare repository [Vir, 2020] that were collected between 2012 and 2020. The 20,000 Android apps in *benign* set are crawled from the official Google Play store. All of the 40,000 apps are submitted to VirusTotal [Vir, 2021], the online scan engines aggregating over 70 anti-virus scanners (including the famous Kaspersky, McAfee, Kingsoft anti-virus engines), to check whether they contains viruses or not. For the *malware* set, we select the malware Android apps that have been labeled by at least five anti-virus engines to ensure their maliciousness, while for the *benign* set, the Android

apps that are not tagged by any anti-virus engines are selected. SEEKER is designed to detect sensor leaks, thus we filter out the Android apps without any onboard sensors by checking whether their code contains the string "`android.hardware.sensor`". The final experimental dataset used in this study consists of 6,724 malware apps and 12,939 benign apps (cf. the 3rd column of Table 3.3). Our experiment runs on a Linux server with Intel(R) Core(TM) i9-9920X CPU @ 3.50GHz and 128GB RAM. The timeout setting for analyzing each app with SEEKER is set 20 minutes.

3.6 Results and Analysis

3.6.1 RQ1 – Feasibility of Detecting Sensor Data Leaks

Our first research question evaluates the feasibility of SEEKER on detecting sensor leaks in Android apps, of which results are illustrated in Table 3.3. For the quantitative aspect, 9,905 potential sensor leaks are identified by SEEKER in 1,596 apps. On average, one Android app could be injected with six sensor leaks. It indicates that the sensor leaks could exist in Android apps which might have been overlooked by the security analysts of Android apps. From the malicious aspect, 14.4% (967 out of 6,724) malware apps are identified with sensor leaks, while 4.9% (629 out of 12,939) benign apps are identified with such leaks. Figure 3.2 further presents the number of sensor leaks detected in each Android app, which shows that each malware app could be identified with more sensor leaks than the benign one. It is significantly confirmed by the Mann-Whitney-Wilcoxon (MWW) test [Fay and Proschan, 2010], of which the resulting *p-value* is less than $\alpha = 0.001$. All of these results imply that malware apps have a higher possibility of containing sensor leaks than benign apps.

Note that: there is lack of the ground-truth dataset about the sensor data leaks in Android apps. To address this limitation, we consider a sensor leak existing in an Android app if there is the data flow interaction between sensor-related sources (i.e., class fields or meth-

Table 3.3: Experimental results of the detected sensor leaks.

Dataset	# apps	# selected apps	# apps identified with sensor leaks	# identified sensor leaks
Malware	20,000	6,724	967	6,103
Benign	20,000	12,939	629	3,802
Total	40,000	19,663	1,596	9,905

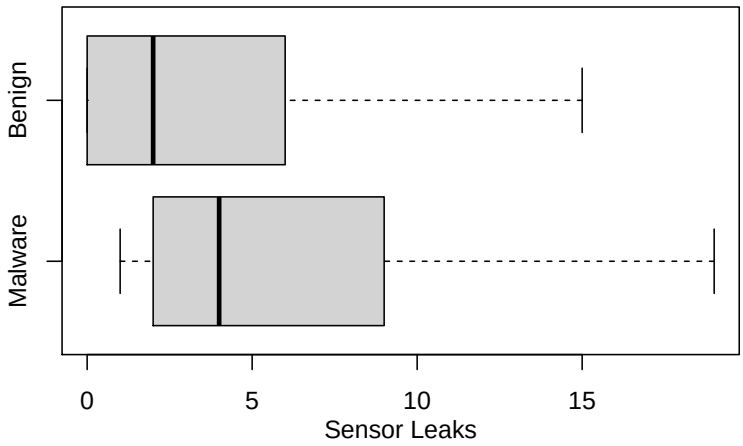


Figure 3.2: Distribution of sensor leaks in each app.

ods) and sinks. With this criterion, we manually checked the 229 sensor leaks detected by SEEKER in 20 randomly selected apps (10 malware apps and 10 benign apps). There are only 4 false-positive identified sensor leaks among the 229 identified sensor leaks in the 20 Android apps, which are caused by inaccurate data-flow analysis results of FlowDroid (we detail this limitation cased by FlowDroid in Section 3.8). Such results show that SEEKER is capable of identifying the sensor leaks in Android apps. Simultaneously, it raises a major alarm for security analysts to pay attention to sensor leaks in Android apps that are not protected by the Android permission mechanism.

RQ1 ☞ Feasibility and Effectiveness

SEEKER is capable of automatically detecting sensor leaks in Android apps. Malware apps present higher possibility of committing sensor leaks than benign apps, and the sensor leaks might be ignored by security analysts.

3.6.2 RQ2 – Characterization of Sensor Leaks

Sources Triggering Sensor Leaks The data leaks in sensor of Android apps are mainly triggered by two kinds of source: field and method (cf. Section 3.4.1). As presented in Table 3.4, ∼80% (= 7941/9905) of identified sensor leaks are triggered by the method sources. For the benign Android apps, ∼85.8% sensor data leaks are sourced from methods, while in the malware Android apps, ∼76.6% leaks are originated from methods. Table 3.5 lists the top-10 most frequent sources triggering sensor leaks identified by SEEKER, which are 8 `getter` methods and 2 public fields from `Sensor`-related classes. We observe that the most frequent leaking source is the method `android.hardware.SensorManager#getDefaultSensor(int)` that is used to get the specific sensor of a given type, which is followed by the field `values` of class `SensorEvent`. The leaking source `android.hardware.SensorManager#getDefaultSensor(int)` occupies ∼89.1% (7074 out of 7941) of the method-triggered sensor leaks, and the field `SensorEvent#values` occupies ∼95.6% (1877 out of 1964) of the field-triggered sensor leaks. The sensor leaks triggered by the two sources occupy ∼90% of all identified sensor leaks.

Table 3.4: Number of identified method/field-triggered sensor leaks.

	# identified method leaks	**# identified field leaks**
Malware	4,677	1,426
Benign	3,264	538
Total	7,941	1,964

Sensor Types of Field-triggered Sensor Leaks The sensor type is essential for deepening the understanding of sensor data leaks, i.e., knowing from which sensor the data is originally collected, as by default, this information is not given in field-triggered sensor leaks (e.g., sourced from the field variable `values` in class `SensorEvent`). The last module of SEEKER is hence dedicated to infer the sensor types of such leaks. Overall, in the 1,964 identified field-triggered sensor leaks, SEEKER successfully infers the corresponding sen-

Table 3.5: Top-10 frequent leaking sources.

Sensor Sources	Malware	Benign	Total
SensorManager#getDefaultSensor(int)	4,326	2,748	7,074
SensorEvent#values	1,358	519	1,877
SensorManager#getSensorList(int)	114	121	235
Sensor#getType()	123	6	129
Sensor#getName()	29	82	111
Sensor#getMaximumRange()	19	73	92
SensorEvent#timestamp	68	19	87
Sensor#getVendor()	12	74	86
Sensor#getVersion()	11	69	80
Sensor#getResolution()	13	64	77

Table 3.6: Top-10 frequent sensor types of field-triggered sensor leaks.

Sensor Type	Malware	Goodware	Total
ACCELEROMETER	1,068	304	1,372
MAGNETIC_FIELD	131	50	181
ORIENTATION	92	84	176
PROXIMITY	12	32	44
LINEAR_ACCELERATION	40	4	44
STEP_COUNTER	14	9	23
TEMPERATURE	12	8	20
GYROSCOPE	7	8	15
PRESSURE	1	13	14
LIGHT	6	5	11

sor types for 1,923 (97.9%) of them. After manually checking the unsuccessful cases, we find that the 41 failed cases are mainly caused by the mistaken usage of sensors which can cause the sensors unexpected functional behavior, such as lacking sensor register information. This high success rate demonstrates the effectiveness of SEEKER in pinpointing the sensor types associated with sensor data leaks.

We further investigate the true-positive rate of the successfully inferred sensor types. Due to the lack of the ground-truth dataset of related sensor types for sensor leaks, we resort to a manual inspection on the source code of 20 randomly selected apps (10 malware apps and 10 benign apps), each of which is identified with at least one field-triggered leak (86 field-triggered sensor leaks in total). All leaks are confirmed with true-positive inferred sensor types, which implies that SEEKER is effective in inferring the sensor types of field-

triggered leaks.

Table 3.6 presents the top 10 leaking sensor types of the identified field-triggered sensor leaks. The type "Accelerometer" is the sensor type of 74.9% and 56.5% of identified field-triggered sensor leaks in the *malware* apps and the *benign* apps, respectively. Android apps widely use the Accelerometer to monitor device motion states by measuring the acceleration applied to a device on three physical axes (i.e., x, y, and z axes). The motion data captured by the Accelerometer can be further processed or analyzed. For example, *Smart-Its Friends* [Holmquist et al., 2001] pairs two devices by acquiring Accelerometer data in a shared wireless medium. Pirttikangas et al. [Pirttikangas et al., 2006] reported that the Accelerometer in smartphones can be used to track the accurate activity of users, such as brushing teeth and sitting while reading newspapers. Such information can also be utilized to steal the PIN of a device through side-channel attacks (such as [Lu et al., 2018] and [Giallanza et al., 2019]).

Apart from the Accelerometer, the other frequent sensor types of field-triggered sensor leaks include MAGNETIC_ FIELD, ORIENTATION, PROXIMITY, LINEAR_ACCELERATION, STEP_COUNTER, TEMPERATURE, GYROSCOPE, PRESSURE and LIGHT. These sensors are also likely to be used to harm the user's privacy. Biedermann et al. [Biedermann et al., 2015] stated that the magnetic field sensor can be exploited to detect what type of operating system is booting up and what application is being started. The orientation sensor can wiretap the device's orientation without requesting any permission, which can be used by attackers to infer the user's PIN. The proximity sensor data can be a trigger to automatically start a phone call recording when users hold the smartphone against their face to make a call. The individual step details can be stored by collecting data from the step counter sensor when the app runs in the background. Temperature, pressure and light sensors are also widely used in IoT devices to monitor environmental conditions, while the gyroscope sensor is utilized to verify the user's identity [Sikder et al., 2021].

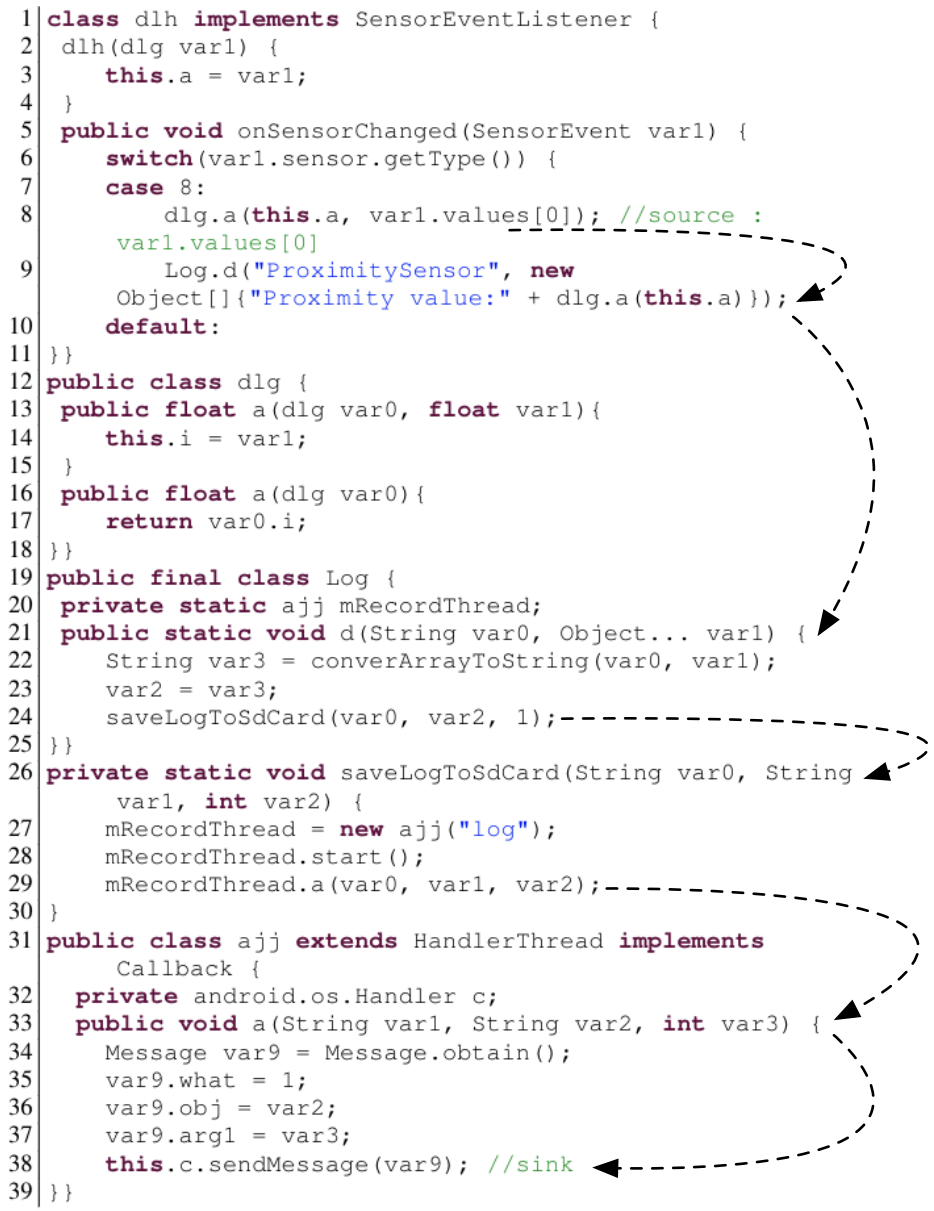
Case Study Here we show two real-world apps that leaks the sensor data, which could be leveraged by attacker to achieve malicious goals.

```
1 final class a.b.b implements SensorEventListener{
2  public void onSensorChanged(SensorEvent var1){
3     float var5 = var1.values[0];
4     float var6 = var1.values[1];
5     float var7 = var1.values[2];
6     Log.v("WindowOrientationListenerN3V", "Raw acceleration vector: x=" + var5 + ",
      y=" var6 + ", z=" + var7);
7 }}
```

Listing 3.2: Example of a sensor leak in *com.n3vgames.android.driver*.

Listing 3.2 showcases a typical sensor leak case in real-world apps. The code snippet is excerpted from a malicious app *com.n3vgames.android.driver*. This app collects raw accelerometer data from the class field `SensorEven#values[]` (lines 3-5), and then leak them through invoking *android.util.Log* API (line 6). The app is flagged as a Trojan that downloads additional executable content from a remote server. While leaking such information may not direct link to its malicious behaviour, it expands the attack surface to the attackers. For example, the sensor information can be used to predict the device's motion state, which may lead to a stealthier attack (e.g., downloading malicious content when the device is not in use). It is worth noting that Zhang et al. [Zhang et al., 2019] have demonstrated the possibility of using the sensor information to launch stealthy attack for taking control of an Android phone via Google's voice assistant.

Figure 3.3 shows another example derived from a phone book app *com.tencent.pb*. It collects the Proximity sensor data (line 8) and eventually send it out through `sendMessage` method (line 41) in a asynchronous thread. The Proximity sensor data was passed as a parameter of method `dlg.a(dlg, float)` (line 8), then the data was stored in the class field `i` of object `dlg`. The data flows through the method *Log.d(String, Object...)* (line 9), which obtains the field variable `i` of object `dlg` as the second parameter via `dlg.a(dlg)` (line 23). Finally, the tainted parameter passed on to the method `Log.saveLogToSdCard(String, String, int)`, which creates a new thread (line 29-33) and send the sensor data out (line 35-42).



```
1  class dlh implements SensorEventListener {
2   dlh(dlg var1) {
3      this.a = var1;
4   }
5   public void onSensorChanged(SensorEvent var1) {
6      switch(var1.sensor.getType()) {
7      case 8:
8          dlg.a(this.a, var1.values[0]); //source :
       var1.values[0]
9          Log.d("ProximitySensor", new
       Object[]{"Proximity value:" + dlg.a(this.a)});
10     default:
11 }}
12 public class dlg {
13  public float a(dlg var0, float var1){
14     this.i = var1;
15  }
16  public float a(dlg var0){
17     return var0.i;
18 }}
19 public final class Log {
20  private static ajj mRecordThread;
21  public static void d(String var0, Object... var1) {
22     String var3 = converArrayToString(var0, var1);
23     var2 = var3;
24     saveLogToSdCard(var0, var2, 1);
25 }}
26 private static void saveLogToSdCard(String var0, String
       var1, int var2) {
27     mRecordThread = new ajj("log");
28     mRecordThread.start();
29     mRecordThread.a(var0, var1, var2);
30 }
31 public class ajj extends HandlerThread implements
       Callback {
32   private android.os.Handler c;
33   public void a(String var1, String var2, int var3) {
34     Message var9 = Message.obtain();
35     var9.what = 1;
36     var9.obj = var2;
37     var9.arg1 = var3;
38     this.c.sendMessage(var9); //sink
39 }}
```

Figure 3.3: Code snippet of sensor value leak excerpted from *com.tencent.pb*.

RQ2 ☞ Characterizing Sensor Leaks

SEEKER is capable of inferring sensor types and pinpointing the corresponding source sensors for data leaks. Our results show that the Accelerometer leaks the most sensor data, both in malware samples and benign apps. The most leaking sources are the method *SensorManager#getDefaultSensor(int)* and the field *SensorEvent#values*, the latter of which has been frequently leveraged in malicious behaviours such as inferring user's PIN.

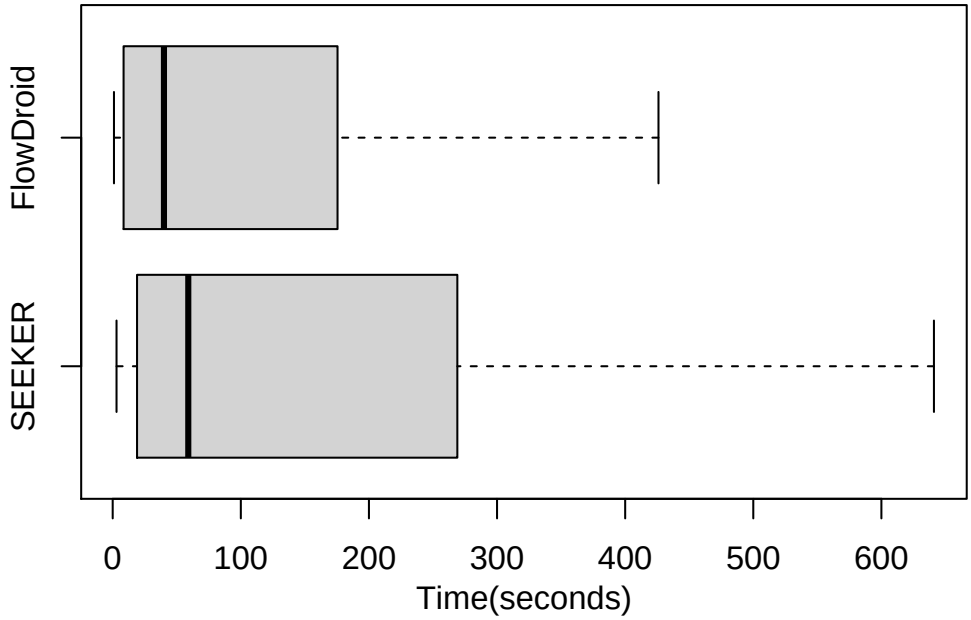


Figure 3.4: Distribution of time Performance spent to analyze an app by FlowDroid and SEEKER, respectively.

3.6.3 RQ3 – Runtime Overhead

SEEKER extends FlowDroid to detect sensor-related data leaks and for inferring the sensor types involved in the leak. We evaluate the runtime overhead of SEEKER and compare it with the original FlowDroid. Figure 3.4 shows the time consumed by FlowDroid and SEEKER, respectively. On average, it takes 177.09 seconds for SEEKER to process an app in our dataset, which is comparable to that of the original FlowDroid (i.e., on average, 132.74 seconds to process an app). As experimentally demonstrated by Avdiienko et al. [Avdiienko et al., 2015], by increasing the capacity of the execution server, the performance of FlowDroid could be further improved. This improvement should also be applicable to SEEKER, making it also possible to analyze real-world apps in practice. The fact that the time difference between SEEKER and FlowDroid is relatively small suggests that it is also capable of applying SEEKER to analyze (in parallel) large-scale Android apps, as what has been experimentally demonstrated to be true for FlowDroid.

RQ3 ☞ Efficiency

The time consumption of SEEKER is acceptable for real-time sensor leak detection, with on average 177.09 seconds for one app without a high increase when comparing with FlowDroid, which is suitable for real-time app analysis.

3.7 Discussion

We now discuss the potential implications and limitations of this work.

3.7.1 Implications

Beyond smartphone apps. The motivating example presented in Section 3.3 is extracted from an attack targeting smartwatches, which also supply sensors to support client apps to implement advanced features. These sensors could be abused by smartwatch app developers, especially malicious attackers. We argue there is also a strong need to characterize sensor leaks in smartwatch apps, not just smart phones. Our preliminary experiment has shown that SEEKER can be directly applied to correctly pinpoint the sensor leaks in the Android-based smartwatch apps that sniff passwords [Chen et al., 2021].

Android has been used on more and more devices, such as TVs, home appliances, fitness machines and cars. The apps in these devices could also all be compromised to leak end-users' sensitive data and hence should also be carefully analyzed before releasing them to the public. SEEKER could also be useful to characterize data leaks for such Android devices and we will examine some of these in our future work.

Beyond sensor leaks. As argued by Zhang et al. [Zhang et al.,], tainted values of string type could be organized as fields in objects, which cannot be detected by state-of-the-art static taint analysis tools such as FlowDroid. This is because FlowDroid only supports methods as sources. Thus, sensitive field sources are overlooked by FlowDroid, giving rise to many false negatives.

Our SEEKER extends FlowDroid to mitigate this research gap by introducing field-

triggered static taint analysis. It is worth highlighting that SEEKER is capable of not only detecting sensor leaks but also pinpointing general privacy leaks, either triggered by source methods or fields. To help users experience this feature, we have committed a pull request to the original FlowDroid on GitHub so that users can easily access Field-triggered Static Taint Analysis by simply configuring their interested field sources in *SourcesAndSinks.txt* file.

Automated approaches for discovering sensitive source fields. In this work, the sensor-related sensitive source fields are identified through manual effort. These are well known to be time-intensive and error-prone. Hence, our current SEEKER approach is not directly applicable for detecting general field-triggered privacy leaks. To achieve this, we need to go through all the fields defined in the Android framework to identify sensitive ones. This is non-trivial as Android is now one of the largest community software projects and contains nearly 10K classes. There is a need to invent new automated approaches to discover sensitive source fields. One possible solution would be to extend the machine learning approach applied in the SUSI tool to support the prediction of sensitive source fields.

3.8 Threats to Validity

Limitation of static analysis. One major limitation of our tool lies in the intrinsic vulnerability of static code analysis when encountering code obfuscation, reflection, native code, etc. These lead to the unsoundness of our approach. However, these challenges are regarded as well known and non-trivial issues to overcome in our research community. In our future work, we want to integrate other useful tools developed by our fellow researchers to overcome these shortcomings. For example, we plan to leverage DrodRA [Sun et al., 2021b, Li et al., 2016b] to reduce the impact of reflective calls on our static analysis approach.

As explained in Section 3.4.3, our sensor type inference approach can not trace the

sensor type in method-triggered leaks when multiple sensors are available on a device. This is because the actual calling object of a method can only be obtained at run-time. We plan to overcome this limitation in our future work by incorporating dynamic analysis approaches to obtain the required run-time values.

Limitations inherited from FlowDroid. Since our SEEKER approach directly extends FlowDroid to detect sensor-triggered privacy leaks, it also has all of the limitations of FlowDroid. For example, FlowDroid may yield unsound results because it may have overlooked certain callback methods involved in the Android lifecycle or incorrectly modelled native methods accessed by the app. FlowDroid is also oblivious to multi-threading and it assumes threads to execute in an arbitrary but sequential order, which may also lead to false results.

Limitations inherited from SUSI. The sensor-related sensitive source methods are collected based on the results of the state-of-the-art tool SUSI. This is also the tool leveraged by the FlowDroid to identify source and sink methods. However, the results of SUSI may not be completely correct – some of its identified sources may not be truly sensitive. However, this threat has no impact on our approach but only on our experimental results. This limitation could be mitigated if a better set of source and sink methods are configured.

Apart from these technical limitations, our work also involves some manual efforts. For example, the sensor-related sensitive source fields are summarized manually by reading the Android developers' documentation. Such manual processes may also introduce errors of their own. To mitigate this threat, the authors of this paper have cross-validated the results, and we release our tool[2] and dataset[3] for public access.

[2]https://github.com/MobileSE/SEEKER
[3]https://zenodo.org/record/4764311#.YJ91jJMzadZ

3.9 Summary

We have presented a novel tool, SEEKER, for characterizing sensor leaks in Android apps. Our experimental results on a large scale of real-world Android apps indicate that our tool is effective in identifying all types of potential sensor leaks in Android apps. Our tool is not only capable of detecting sensor leaks, but also pinpointing general privacy leaks that are triggered by class fields. Although there are related works on sensor usage analysis, to the best of our knowledge, there is no other work that thoroughly analyses Android sensor leakage. Unlike previous works, our tool is the first one to characterize all kinds of sensor leaks in Android apps. We extend FlowDroid for supporting field sources detection (i.e., merged to FlowDroid via pull #385 on Github[Xiaoyu Sun, 2021]), which we believe could be adapted to analyze other sensitive field-triggered leaks. To benefit our fellow researchers and practitioners towards achieving this, we have made our approach open source at the following Github site.

`https://github.com/MobileSE/SEEKER`

Chapter 4

Demystifying Hidden Sensitive Operations in Android apps

Xiaoyu Sun, Xiao Chen, Li Li, Haipeng Cai, John Grundy, Jordan Samhi, Tegawendé F. Bissyandé and Jacques Klein, 2022, Demystifying Hidden Sensitive Operations in Android apps. ACM Transactions on Software Engineering and Methodology (TOSEM) 2022. https://arxiv.org/abs/2210.10997

4.1 Introduction

Android is the most adopted mobile operating systems in terms of users, applications and developers [Anthony Scarsella, 2020]. However, its popularity means that legitimate developers must co-exist with malware writers. Reports on many different kinds of attacks are presented in the technology and lay media. For example, security researchers have reported a malicious "clicker trojan"[1] which has been bundled into 34 different Google Play apps that have already been installed more than 100 million times[2]. On a larger scale, antivirus engines have been flagging a large number of apps as potential threats. For example, as of October 2020, the popular AndroZoo dataset [Allix et al., 2016] has recorded more than 226,000 Android GooglePlay apps than have been flagged as adware/malware by at least 5 Antivirus products, and this number is still growing. Those adware/malware often not work along but collaborate with many third parties over the internet. Some of the representative malicious behaviors include leading users to malicious websites through devious

[1]Such as the *Android.Click.312.origin* trojan and its modified variant Android.Click.313.origin trojan. This aims to generate fraudulent click-through and subscription revenues.

[2]https://www.forbes.com/sites/zakdoffman/2019/08/13/android-warning-100m-users-have-installed-dangerous-new-malware-from-google-play/#1956f51c22a9

advertisements [Liu et al., 2020b, Xu et al., 2020, Dong et al., 2018a, Dong et al., 2018b], distributing malicious apps in the mobile network through drive-by downloads [Cova et al., 2010], leaking users' sensitive data to web servers through HTTP connections [Sun et al., 2021a, Liu et al., 2021b, Li et al., 2015a, Gao et al., 2020], etc.

To protect Android users against the rapid spread of malware, the research and practice communities have implemented a variety of measures and proposed several approaches to detect malware [Arp et al., 2014, Mariconti et al., 2016, Zhao et al., 2021, Li et al., 2017b, Sun et al., 2022, Xu et al., 2022]. These include static code analysis-based approaches [Li et al., 2017a, Li et al., 2019], dynamic testing based approaches [Kong et al., 2018], and learning-based approaches [Liu et al., 2022b]. Unfortunately the emergence of many different malware detection techniques has also stimulated malware attackers into being more innovative to increasingly better hide malicious behaviour, in order to bypass static code analysis (e.g., via obfuscation) and even dynamic detection (e.g., sensing of sandbox execution). In practice, sophisticated code obfuscation techniques [Moser et al., 2007] are being leveraged by attackers to hide their malicious program behavior, leading to false negatives in most static analyses thus resulting in imprecise and unsound results. Camouflage techniques have been frequently leveraged by attackers to evade dynamic testing approaches [Rasthofer et al., 2017, Egele et al., 2008]. Attackers often introduce a so-called logic bomb or time bomb to set off malicious functions only after certain conditions are met. For instance, after knowing that Google applies a dynamic analysis tool called *bouncer* to scan every app submitted to Google Play for five minutes, as revealed by Oberheide et al. [Oberheide and Miller, 2012], a bunch of malicious apps has been created and been demonstrated to be capable of penetrating Google's bouncer vetting system by simply waiting five minutes before triggering their malicious behavior.

To cope with such hidden malicious behaviors, researchers have devised new detection approaches. For example, Fratantonio et al. [Fratantonio et al., 2016] have proposed an

approach called TriggerScope to detect hidden behaviors triggered by predefined circumstances such as events related to location, time, and SMS. However, TriggerScope is not capable of detecting such malicious activities hidden behind other trigger types, such as the existence of other services (i.e., other than location, time and SMS). In line with this research, Pan et al. [Pan et al., 2017] have proposed a machine learning-based approach aiming to discover unknown trigger types. Their approach, however, needs to manually label a dataset for training, which is known to be resource-intensive and error-prone.

Static analyzers suffer less than dynamic approaches from evasion techniques such as logic bomb or time bomb. In particular, regarding sensitive flow detection (also called privacy leak detection), numerous static analysis tools have been proposed such as FlowDroid [Arzt et al., 2014] (and its extension IccTA [Li et al., 2015c]), AMANDROID [Wei et al., 2014], or DroidSafe [Gordon et al., 2015]. Although these tools are able to track sensitive flows (which are often hidden) by bringing key new contributions to the research community, they still face some well-known limitations [Qiu et al., 2018]: their inherent over-approximations inevitably lead to false alarms, which, for some analyzers, occur at a high rate, making them impractical. Consequently, when building on static analysis, manual investigation is often required. Unfortunately, such efforts cannot scale. Dynamic validation then appears as an alternative. Unfortunately, runtime execution often misses hidden sensitive flows due to the implementation of evasion techniques by attackers. While some effort (e.g., [Fratantonio et al., 2016, Pan et al., 2017]) has been put to *characterize* Hidden Sensitive Operations (HSOs) in Android apps, our community has not yet proposed dedicated approaches to *detect and explain* such operations, allowing attackers to achieve malicious behaviors while bypassing certain security vetting mechanisms.

We fill this research gap in this work by proposing a new prototype tool, HiSenDroid, which deploys an automated static app analyzer tailored for detecting *hidden* sensitive operations. HiSenDroid performs a sequence of static analyses, including call graph analysis,

forward data-flow analysis, inter-procedural backward data-flow analysis, etc. For exposed HSOs, HiSenDroid further goes one step deeper to record detailed information for explaining why these HSOs should be flagged as such.

To summarize, key contributions of our work include:

- We propose using a static analysis approach to discover hidden sensitive operations that are not exposed to the state-of-the-art static and dynamic analysis tools in Android apps. To this end, we leverage control flow and data flow analyses to identify the unique code level characteristics of hidden sensitive operations.

- We designed and implemented a prototype tool HiSenDroid for analyzing hidden sensitive operations. We release HiSenDroid as an open source project [Sun, 2022] for supporting security analysts in their analysis needs and fostering further researches in this direction.

- We evaluated HiSenDroid on a large-scale dataset that contains 10,000 benign and 10,000 malware samples, and discovered emerging anti-analysis techniques employed by malware samples, such as fulfilling certain restrictions related to *time*, *location*, *SMS message*, *system properties*, *package manager*, and other logics.

- With the help of FlowDroid [Arzt et al., 2014], a static taint analyzer, we further experimentally show that HSOs have been recurrently leveraged by attackers to leak sensitive user information.

The rest of the study is organized as follows: Section 4.2 defines HSO and presents the motivation of our research, i.e., why there is a strong need to demystify HSO. Section 4.3 depicts the design and implementation of the proposed approach. Section 4.4 and Section 4.5 respectively describe the characteristics of common and susipious HSOs detected by our approach from a large-scale dataset. Section 4.6 presents a practical implication of

our approach by characterizing sensitive data leaks triggered by HSOs. Section 4.7 discusses the limitations of the tool. Section 4.8 reviews the related works, and finally Section 4.9 concludes this paper.

4.2 HSO Definition and Motivation

We conducted an exploratory study to understand the characteristics of *Hidden Sensitive Operations* (HSO) in Android apps. We first dumped operations in a set of real-world Android malware. Then, we manually examined those operations to observe the characteristics of such operations that could be considered as hidden-triggered operations. Based on our manual summarization, we found that (1) *if statement* and the notion of *branch* are key in the definition of HSO; (2) the *if statement* contains a specific *operation* that triggers the hidden sensitive flows, and this trigger condition is related to Android API.

Let B denote one of the two branches of an *if-then-else statement*, or the branch of an *if statement* where the *else* branch is considered empty.

Definition 1 [Hidden Sensitive Branch (HSB)]: B is an HSB if it fulfills the following **rules**:

1. B contains sensitive Android APIs, and these APIs are different from those contained in the other branch involved in the *if-then-else statement*. The rationale behind this condition is that a hidden branch is supposed to achieve some sensitive behaviors that are different from those of the "normal" branch (i.e., non-HSB), which per se might also access sensitive APIs as part of the app's expected behaviors.

2. B does not involve any of the variables appearing in the *condition expression* of the *if-then-else statement*. The rationale behind this is that the branch is triggered by conditions that are also different from its (sensitive) behaviors.

Less formally, an HSB could be defined as an "if branch" which accesses sensitive

APIs, and which is fully "independent" of the *if condition* and the other branch of the *if statement*.

Let C denote the $condition$ of an *if statement*.

Definition 2 [Hidden Sensitive Operation (HSO)]: An HSO is an HSB that is triggered by a condition C containing values obtained via (or directly impacted by) Android system APIs or system properties (i.e., attributes of system classes). This may return different values when being executed under different circumstances, so as to triggering hidden sensitive operations.

```
1  public class MainActivity extends AppCompatActivity {
2   protected void onCreate(Bundle savedInstanceState) {
3    SmsManager smsManager = SmsManager.getDefault();
4    ED ed = new ED(this);
5    StringBuilder message = new StringBuilder();
6
7    if(ed.checkPackageName()) {
8     TelephonyManager tm = (TelephonyManager)
       getSystemService(Context.TELEPHONY_SERVICE);
9     String imei = tm.getDeviceId();
10    String phoneNumber = tm.getLine1Number();
11    String subscriberId = tm.getSubscriberId();
12    message.append(imei);
13    message.append(phoneNumber);
14    message.append(subscriberId);
15    smsManager.sendDataMessage("+115800763861", null, (short)1001,
       message.toString().getBytes(), null, null);
16   } else {
17    //benign string operations
18 }}
19
20 public class ED {
21  public ED(Context pContext) {
22   mContext = pContext;
23   mListPackageName.add("com.google.android...genymotion");
24   mListPackageName.add("com.bluestacks");
25   mListPackageName.add("com.bignox.app");
26  }
27  public boolean checkPackageName() {
28   if (!isCheckPackage || mListPackageName.isEmpty()) {
29    return false;
30   }
31   final PackageManager packageManager = mContext.getPackageManager();
32   for (final String pkgName : mListPackageName) {
33    final Intent tryIntent = packageManager.getLaunchIntentForPackage(pkgName);
34    if (tryIntent != null) {
35     final List<ResolveInfo> resolveInfos =
       packageManager.queryIntentActivities(tryIntent,
       PackageManager.MATCH_DEFAULT_ONLY);
36     if (!resolveInfos.isEmpty()) {
37      return true;
38     }
39    }
40   }
41   return false;
42 }
```

Listing 4.1: An example of a real-world hidden sensitive data flow.

Listing 1 exemplifies a simplified code snippet illustrating these definitions in practice. Note that Listing 1 presents the typical characteristics of an HSO in many real-world apps that we have manually analyzed. At line 7, the app firstly checks if it is running on one of the popular Android emulators (i.e., *genymotion, bluestacks, and bignox*). If not, the app reads the device information and sends it to a hard-coded phone number through an SMS. Otherwise, if an emulator environment is detected, it will only perform some unharmful string operations (ignored). In this example, three private data – namely the device's IMEI, IMSI, and phone number – are retrieved in lines 9-11 and sent to a hard-coded phone number via SMS (line 15). All of these three leaks are hidden behind the trigger condition *ed.checkPackageName()* (line 7). The trigger condition checks the return value of a self-defined method *checkPackageName()* (line 30), which is determined by several other *if-conditions* defined in the invoking method (lines 31,37,39). Finally, the trigger condition in the HSO is traced back to a system API *PackageManager.queryIntentActivities()* (line 38) (**cf. Definition 2**). This trigger condition examines whether popular Android emulator packages (lines 26-28) are available in the device, i.e., checking if the app is running on these emulators. If the running environment is not one of the hard-coded emulators, the HSO will be performed. Otherwise, benign string operations are executed (lines 17-19) (**cf. Definition 1**).

4.3 Approach

To better help security analysts understand Hidden Sensitive Operations (HSO) placed in Android apps, we designed and implemented a prototype tool, named HiSenDroid, to automatically locate such operations in Android apps. HiSenDroid takes as input an Android app and outputs a set of hidden sensitive operations. Fig. 4.1 illustrates the working process of HiSenDroid. It achieves the aforementioned goal through three main modules, namely: (1) Hidden Sensitive Branch Location; (2) Trigger Condition Inference; (3) Suspicious

HSO Detection and Explanation. We now respectively detail these three modules.

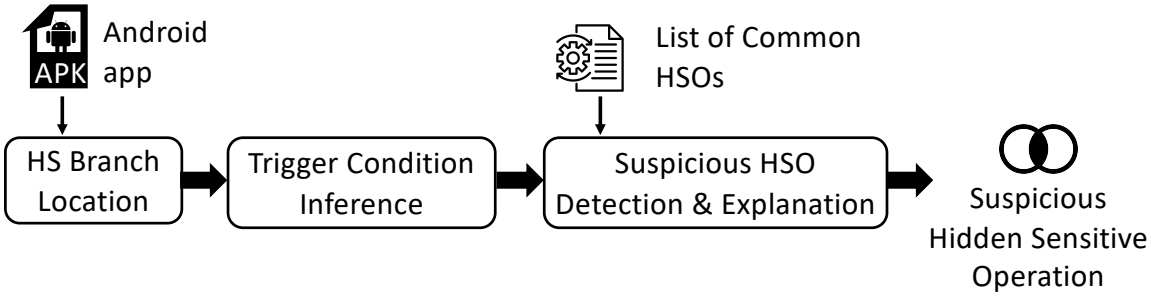


Figure 4.1: The working process of HiSenDroid.

4.3.1 Hidden Sensitive Branch Location

The first module of HiSenDroid is responsible for locating hidden sensitive branches (HSBs) in Android apps (i.e., fulfilling the rules in Definition 1). Towards locating HSBs, this module first statically goes through all the methods that appeared in the DEX file of the input APK. For each method, this module then constructs an intra-procedural control-flow graph (CFG) and traverses the graph to locate *if-then-else statements*. Once an *if-then-else statement* is located, it further extracts the sensitive APIs accessed by the two branches (hereinafter referred to as *if-branch* and *else-branch*). Sensitive APIs are such methods that are protected by Android permissions, which are classified following the latest Android API-permission mappings PSCout [Au et al., 2012], Axplorer [Backes et al., 2016], Arcade [Aafer et al., 2018], and NatiDroid [Li et al., 2022a]. Any of the two branches will be considered a potential HSO if it has indeed accessed sensitive APIs that are different from the APIs accessed by the other branch.

When extracting sensitive APIs, in order to obtain a *soundy* result [Livshits et al., 2015] (e.g., including all the sensitive APIs accessed by a potential HSB), this module traverses not only the methods directly presented in the HSB but also all the methods that could be reached from the branch. This process is made possible by first constructing a call graph (CG) for the input APK. Unfortunately, as discussed by many existing works, Android

apps do not have a single entry point (e.g., *main()*) that connects other parts of the application code, making static analyses challenging to cover all the app code. Fortunately, this challenge has been well addressed by the state-of-the-art by artificially creating a so-called dummy main method, connecting together all the separated code parts, including system-driven lifecycle methods and event-driven callback methods [Arzt et al., 2014]. Based on our observation and the findings of previous work [Pan et al., 2017], the connection between trigger conditions and the operations along its paths is often weak. Indeed, the variables appearing in triggers typically do not propagate data flow to its following paths. Take Listing 1 as an example, the app checks if it is running inside Android emulators at line 7, where the trigger condition code itself is not supposed to steal private data and is only meant to determine the right situation for running hidden code. To leverage this property, we attempt to check whether variables appearing in the HSB have data dependency with any variable within the condition expression. Thus, for a given potential HSB, this module goes one step further to check if any of the variables appeared in the HSB's *condition expression* has been leveraged by the HSB code. If so, this HSB will not be considered as a true HSB and thereby will be excluded from further analyses. This module achieves this by conducting a simple intra-procedural control-flow analysis. In a case of true HSB, there should not be intersections between the set of variables that appeared in its *condition expression* and those within the branch.

4.3.2 Trigger Condition Inference

After locating HSBs, the second module goes one step deeper to infer hidden sensitive operations (HSOs) so as to fulfill Definition 2. Given a true HSB, the idea of detecting HSOs is to infer the detailed trigger conditions that lead to the execution of the HSB.

We began with a preliminary study to understand what kinds of trigger conditions have been used to hide suspicious APIs, as identified from the literature [Pan et al., 2017, Pet-

sas et al., 2014, Chen et al., 2008, Vidas and Christin, 2014, Jing et al., 2014, Diao et al., 2016, Costamagna et al., 2018, And, , Hac, , tam, , Norboev et al., 2017] on trigger conditions. For example, Petsas et al. [Petsas et al., 2014] investigated anti-analysis techniques that can be employed by Android apps to evade detection, including pre-initialized static information(e.g., IMEI value), dynamic information that does not change (e.g., Sensors data) and VM instruction emulation (e.g., hardware variable). In their paper, they demonstrated how dynamic analysis could be evaded by the aforementioned trigger conditions in an emulated environment. Pan et al. [Pan et al., 2017] further summarize that almost all the trigger conditions of HSOs can be characterized by **System Properties** (e.g., OS or hardware traces of a mobile device) or **Environment Parameters** (time, locations, SMS, etc.). To the best of our knowledge, the values in both types can be obtained through Android system APIs. In other words, an HSO trigger condition is expected to involve, directly or indirectly, one or more system API calls for interacting with the Android operation system. Therefore, since it is very important to identify all possible trigger conditions, we propose considering all the condition checks to infer HSO's trigger conditions[3] as long as they involve system properties, environment parameters, and any other values yielded by system APIs.

In this work, we follow the same criteria to infer HSOs (i.e., the trigger conditions involves values obtained through Android system APIs). Specifically, to infer the trigger conditions, for each of the variables that appeared in the HSB's *condition expression*, there is a need to conduct backward data-flow analyses to locate its definition statement. The code block between the definition statement and the *if-then-else statement* is then referred to as a Condition Triggering Block (CTB). Then, given a potential HSO, we check whether

[3]We remind the readers that state-of-the-art studies (e.g., by Moser et al. [Moser et al., 2007] and Zeng et al. [Zeng et al., 2018]) have further revealed that obfuscation (via reflective calls or opaque predicates) could be leveraged to complicate the inference of trigger conditions (e.g., changing the way how a system property is obtained from the system). We do not take obfuscation as a type of trigger condition but will only consider it as a technique that complicates the process of identifying trigger conditions. We will discuss the impact of obfuscation on our approach at the end of Section 4.5.

a system API is involved in the *definition statement* of the HSO's CTB. If not, we will regard this HSO as a false result and consequently will not consider it for further analyses.

When inferring the *definition statement*, inter-procedural analysis needs to be taken into account because the trigger conditions can be defined in other methods and transferred to the HSB via callee's returned values or caller's parameter values. Indeed, take the code snippet shown in Listing 4.1 as an example, the trigger condition is actually defined in method *checkPackageName()* despite the HSB is seated in the *onCreate()* method. Fig. 4.2 illustrates the backward tracking flow showing how our approach identifies the trigger condition. When there is a method involved in the backward tracking flow, our data-flow analysis will keep tracking the method's caller object as it may be relevant to the definition of the trigger condition. For example, our analysis will keep tracking $r1 when statement *$r1.isEmpty()* is reached. If the method is a user-defined function, our data-flow analysis will further jump into the method and keep tracking its returned variables (all variables will be tracked if there are several return statements). The backward data-flow analysis will terminate if System APIs are identified, or Android's entry-point methods (such as UI callback methods or components' lifecycle methods) are reached. The analysis will also stop if the condition is linked to a constant value that is further not originated by *if-statements*.

4.3.3 Suspicious HSO Detection

The last module takes the outputs of the previous module to detect hidden sensitive operations, following the rules presented in Definition 1 and Definition 2 (cf. Section 4.2). Unfortunately, these rules are not perfect and may introduce false-positive results that have similar characteristics of HSOs but are actually user intended behaviors. Indeed, for the same operations, under different circumstances, they could be flagged as conventional usages or suspicious operations and could lead to benign or user intended malicious behaviors. These false results include common programming patterns used in legitimate *if-then-*

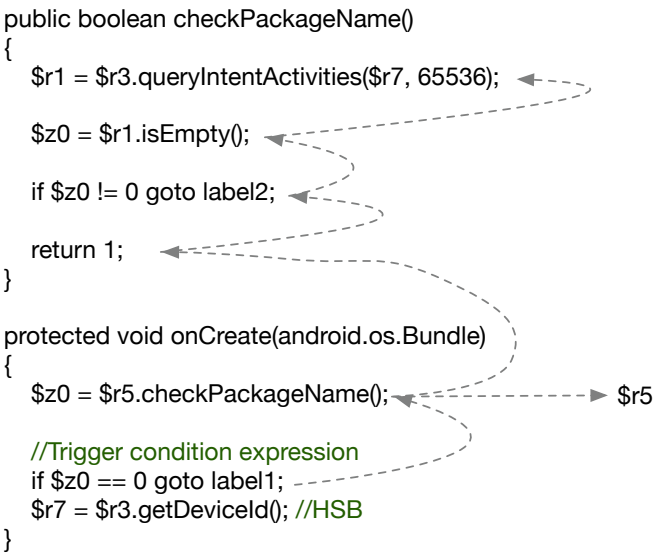


Figure 4.2: The simplified working process of the trigger condition inference module. The code is presented in simplified Jimple, which is an intermediate representation of Soot [Lam et al., 2011]. Soot is the underline static analysis framework leveraged by HiSenDroid to achieve the backward data-flow analysis.

else statements (hereinafter referred to as *conventional usages*), which should be excluded by HiSenDroid . Therefore, we resort to building a list of conventional usages (or whitelist) and based on it, in the last module of HiSenDroid, we filter out non-malicious HSOs and only keep suspicious HSOs.

Nevertheless, we argue that it is non-trivial to understand the developer's intention behind the operations. Therefore, in this last module, in addition to automatically detect suspicious hidden sensitive operations, HiSenDroid goes one step deeper to also provide adequate details to explain why an suspicious HSO is flagged as such, i.e., what is the trigger condition, what is the logic of the *if condition*, and what are the sensitive behaviors triggered if the logic is fulfilled. This function is provided for helping security analysts understand whether the flagged HSOs should be regarded as malicious or not.

By leveraging HiSenDroid, in Section 4.4, we study and collect *conventional usages* in large sets of Android apps, whereas in Section 4.5, we put the emphasize on *suspicious* HSOs.

4.4 Conventional Usage

The overall goal of this work is to detect hidden sensitive operations so as to unveil the evasive technologies that are frequently leveraged to hide malicious behaviors. In this section, we evaluate our approach based on a large set of Android apps towards checking if our approach HiSenDroid is capable of fulfilling this goal. Specifically, in this section, we conduct an exploratory study of recent hidden sensitive operations aiming to understand the current status quo of conventional usages and build a comprehensive list of conventional usages (to be used by HiSenDroid to discriminate suspicious HSOs from conventional usages).

Recall that our approach, in its last working step, takes as input a customizable list of conventional usages to filter out non-suspicious HSOs, which subsequently helps in saving significant security analysts' efforts as they now only need to scrutinize the retained small number of likely suspicious HSOs. Towards identifying such conventional usages, we apply a semi-automatic process to summarize based on their frequency of occurrence. The conventional usage whitelist is built based on reasonable assumptions that legitimate HSOs frequently appear in Android apps, including both malware[4] and goodware. We manually inspected the trigger APIs that have appeared more than 50 times in our dataset and determined if they should be categorized as a conventional usage. By doing so, we defined seven major categories of conventional usages. Also, the results of our manual analysis are cross-validated by two authors. The two authors first independently conduct the manual analysis (to discover knowledge with support evidence from various software artifacts). They then had physical meetings to discuss, merge, and finalize the results.

Experimental Setup. We applied HiSenDroid (with the list of conventional usages set to be empty[5]) on a dataset that contains 10,000 malware samples (referred to as *malware*

[4]Malware is included because often not all of its code is malicious. It might contain a malicious payload but the other code could still remain benign.

[5]The experimental results should contain both conventional usages and suspicious HSOs.

set) and 10,000 benign apps (referred to as *benign set*). The *malware set* was collected from VirusShare [Vir, 2020] from 2012 to 2020. To better reflecting recent trends on the deceptive techniques used in malware samples, we only include the samples whose first seen date was on or after 2016. The malware samples were submitted to VirusTotal[6] for screening, and only the ones that have been labelled by more than five anti-virus engines (VirusTotal has hosted over 70 anti-virus scanners) were selected.

The *benign set* was randomly selected from a pool of more than 100,000 apps crawled from Google Play in 2019, which are further scanned to ensure non of them are tagged by VirusTotal.

Our tool has identified 45,342 HSOs (35,974 in the *malware set*, and 9,368 in the *benign set*) triggered by 54,152 conditions. Note that some HSOs may be triggered by more than one condition (e.g., multiple conditions in a CTB that are connected by *AND* or *OR* operators). Towards evaluating the precision of HiSenDroid, i.e., the identified HSOs meet our previous rule definitions, we manually examine 20 randomly selected APKs from the total 8,107 apps that have been identified to contain at least one HSO. From these apps, our approach identified 157(with a confidence level of 95% and a confidence interval of 7.81%) HSOs in total, among which 155 of them are eventually confirmed to be true HSOs, giving an precision of 98.7%. This result suggests that HiSenDroid is capable of identifying HSOs in Android APIs.

Figure 4.3 further presents the distribution of the number of HSOs detected in the apps from *benign set* and *malware set*. Expectedly, malware samples involve significantly more HSOs than that of benign apps, as confirmed by the *p-value* a Mann-Whitney-Wilcoxon (MWW) test at a significance level[7] at 0.001 [Fay and Proschan, 2010]. This result suggests that HSOs are more favored by malware than benign apps. Hence, our community

[6]https://www.virustotal.com

[7]Given a significance level $\alpha = 0.001$, if p-value $< \alpha$, there is one chance in a thousand that the difference between the two datasets is sue to a coincidence.

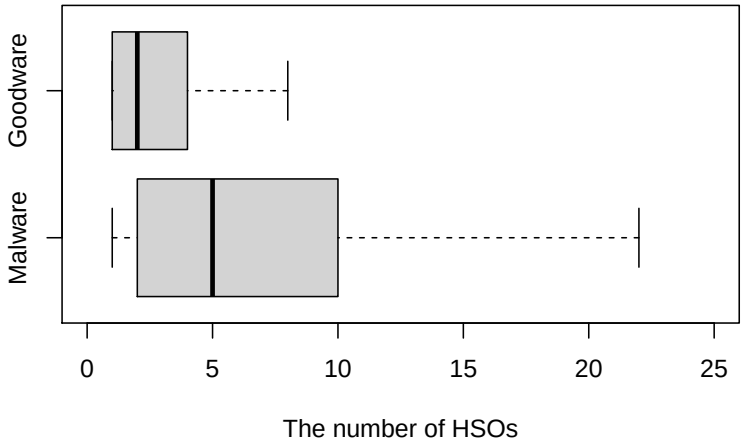


Figure 4.3: Distribution of the number of HSOs in *benign set* and *malware set*.

should pay more attention to the appearance of HSOs to help security analysts better dissect malicious apps.

Based on the previous experimental results, we manually analyzed the trigger conditions and the corresponding hidden operations to identify *conventional usages*, i.e., HSOs (at least based on our definition) that are actually legitimate and occur relatively often in Android apps. We first inspected the trigger APIs that have appeared more than 50 times in our dataset and determined if it is a *conventional usage*. By doing so, we identified seven major categories of *conventional usages*. Then we reviewed each of the rest of the cases to further filter out the other *conventional usages*. Finally, 43,141 *conventional usages* have been identified, out of which 40,412 cases belong to the seven major categories. As the whitelist is generated by manual analysis, it cannot cover all possible *conventional usages*. However, we believe that the majority of the *conventional usages* have been identified (i.e., from the seven categories), new special cases can always be added to the list and incorporated into HiSenDroid in the future. We now elaborate on the seven major categories, each with an example code snippet presented in Listing 4.2.

```
1 // conventional usages - SDK version check
2 public void addJavascriptInterface(Object var1, String var2) {
3  if (VERSION.SDK_INT < 17) {
4   TaoLog.e("HybridWebView", "addJavascriptInterface is disabled before API level 17
      for security.");
5  } else {
6   super.addJavascriptInterface(var1, var2);
7 }}
```

```
8
9  // conventional usages - User Interface
10 class ClickEvent implements View.OnClickListener {
11     public void onClick(View view) {
12         if(view.getId() == backButton.getId()){
13             webView.goBack()
14         }
15         else if (view.getId() == reloadButton.getId()){
16             webView.reload();
17 }}}
18
19 // conventional usages - File Handling
20 public static File inputstreamtofile(InputStream ins) {
21  File SDFile = Environment.getExternalStorageDirectory();
22  File desDir=new File(SDFile.getAbsolutePath());
23  File newFile=new File(desDir.getAbsolutePath() + File.separatorChar+"myPaint.png") ;
24  if(desDir.exists()){
25   OutputStream os = new FileOutputStream(newFile);
26   while ((bytesRead = ins.read(buffer, 0, 8192)) != -1) {
27    os.write(buffer, 0, bytesRead);
28 }}}
29 // conventional usages - Permission Check
30 public static String getDeviceInfo(Context context) {
31  if (checkPermission(context, Manifest.permission.READ_PHONE_STATE)) {
32   String device_id = tm.getDeviceId();
33  } else {
34   requestPermissions(context, new String[] {Manifest.permission.READ_PHONE_STATE},
      REQUEST_CODE)
35 }}
36
37 // conventional usages - Network
38 public String g() {
39 var1 = ((ConnectivityManager)a.getSystemService("connectivity"))
       .getActiveNetworkInfo();
40 var9 = var1.getType();
41 if(var9 == 1){
42  var10 = ((WifiManager)a.getSystemService("wifi")).getDhcpInfo();
43 }}
44
45 // conventional usages - Intent Management
46 public void onReceive(final Context context, Intent intent) {
47  String action = intent.getAction();
48  if (action.equalsIgnoreCase ("android.net.conn.CONNECTIVITY_CHANGE") {
49   connectivityManager.getActiveNetworkInfo();
50 }}
51
52 // conventional usages - SharedPreferences
53 public class VpnAddressIp{
54  public SharedPreferences sp;
55  public String VPNAddress() {
56   sp = context.getSharedPreferences("SP", Context.MODE_PRIVATE);
57   VPNflag = sp.getInt("VPNFlag", 1);
58   VPNAddress vpnaddress = new VPNAddress(context);
59   if (VPNflag == 1) {
60    VPNAddressBean bean = vpnaddress.queryVPN(1);
61    networkaddress = bean.getNetwork();
62   }
63   return networkaddress;
64 }}
```

Listing 4.2: Examples of conventional usages.

SDK Version. With Android system update, new APIs are defined to replace old ones. To maintain the compatibility of apps across different Android versions, it is a common

practice to check the SDK version before deciding the right API to use. Lines 2~7 show a simplified code snippet of a legitimate conventional usage that fulfills all the rules we defined for an HSO. The code checks whether the Android version is newer than *Android level 17* (line 3), if so, the app leverages the *addJavascriptInterface()* API to inject Javascript into the *WebView* (line 6), otherwise, it logs an error message (line 4) as the API is not available in the Android version lower than 17. While SDK version check commonly exists in both malware and benign apps (with 10,303 and 3,223 cases, respectively), this check does not intend to hide the behaviors within the *if-then-else* statement and hence should be excluded from the HSO results.

User Interface. When the user interacts with UI widgets (e.g., press a button), it retrieves and compares the UI widget's *id* (i.e., a system API) to determine which widget has been fired. If there happened to be a sensitive API invoked in one of the branches' statements, this code block will be misidentified as HSO. Lines 10~17 show an example of a button's callback method, which checks the ID of the buttons (lines 12,15) and either go back to the previous webpage (line 13) or reload the current page (line 16). User Interface has 8,052 instances in the *malware set* and 2,426 instances in the *benign set*.

File. The existence of a file or a directory is usually checked before file operations, such as reading and writing files. The code for checking file existence typically put the subsequent actions in one branch (where the file does exist), and show an error message in the other branch (where the file does not exist). In some cases, it even has only the *if-branch*. Therefore, it satisfies the rules mentioned above and will be mistakenly identified as an HSO. Our results have observed 7,217 and 1,701 cases in our *malware set* and *benign set*, respectively. A file checking example can be found in lines 20~28, where the code checks the existence of an external storage (line 24), and copy an image there (lines 26,27).

Permission. Since Android 6.0, the dangerous-level permissions need to be explicitly checked and requested before accessing the APIs protected by these permissions. The

example code for checking permission can be found in lines 31~36. It first checks whether the app has been granted *READ_PHONE_STATE* permission (line 32). Then, the app either invoke the permission protected API (line 33) or request the missing permission (line 35) based on the check result. Even though a sensitive API *getDeviceId()* is called in one branch, which behaves quite differently than the other branch, it does not mean to hide this behavior. Therefore, it is regarded as a *conventional usage*. Permission check has appeared 6,727 and 936 times in the HSOs identified in the *malware set* and *benign set*, respectively.

Network. Network information (e.g., network type, connection status, etc.) is always checked before performing network-related behaviors, ensuring that the network status is suitable for accomplishing the subsequent tasks. For example, the network type (e.g., WiFi, cellular, etc.) is checked before downloading large files, and if it is on the cellular network, the download will be suspended. Another example demonstrated in lines 39~44 examines the type of connected network (line 41) and get its DHCP information if the phone is connected to WiFi (1 is the value of ConnectivityManager#TYPE_WIFI). There are 5,224 an 744 identified HSO cases that are related to the *Network* in the *malware set* and the *benign set*, respectively.

Intent. *Intent* is a crucial mechanism to assist the communication between different components in the Android system. *Intent* has various legitimate usages, including starting activities and services, passing data and properties, etc. Lines 47~51 demonstrate a legitimate example of handling the callback method of receiving an *Intent*. In this example, it checks the *action* defined in the received *Intent*, and calls *getActiveNetworkInfo()* method (i.e., a sensitive API) if the *action* is *CONNECTIVITY_CHANGE*. There are 1,911 an 1,011 identified HSO cases that are related to the *Intent* in the *malware set* and the *benign set*, respectively.

SharedPreferences. In Android system, data can be saved as <key, value> pairs and stores as a *SharedPreferences* object in a file that can be accessed by *getSharedPrefer-*

ences() interface. It provides a lightweight and easy-access data store mechanism, which is widely used in storing small collection of data, such as configurations of the app. Reading the values from the *SharedPreferences* and action accordingly is considered a legitimate behavior. Lines 54~65 illustrate an example of using *SharedPreferences*, where the code retrieves the value of a configuration item "VPNFlag" (line 58) from a *SharedPreferences* object named "SP" (line 57), and query the corresponding VPN services accordingly (lines 60~62). There are 878 and 358 cases involving the usage of *SharedPreferences* in our *malware set* and *benign set*, respectively.

Completeness of conventional usages. Since the conventional usage categories are summarized with manual efforts on a given set of apps, they may not be representative and thereby may not cover all possible cases. Therefore, in this work, we go one step deeper to further investigate the completeness of all the seven categories of conventional usages by applying our approach to another set of randomly selected 10,000 malware and 10,000 benign apps from AndroZoo [Liu et al., 2020a]. We remind the readers that AndroZoo includes over 10 million Android apps that were collected from both the official Google Play store and several third-party app markets. To avoid potential biases in our results, we made additional efforts to remove potentially duplicated apps (i.e., different versions of the same app), and only the latest version is retained. For the 20,000 apps, we apply HiSenDroid to analyze these apps and inspect the trigger conditions that have appeared more than 50 times. We then manually determine if they are conventional usage. To do this, two of the authors spent ten person-days manually summarizing conventional usages (e.g., API-API or Key-API pairs). After manually checking the experimental results and the bytecode of apps, we have totally picked up 41,035 conventional usages, among which 38,920 cases fall into the predefined whitelist (with a success rate of 94.8%). This result shows that, despite testing on different apps, our whitelist is still quite stable and effective in eliminating conventional usages.

Apart from the aforementioned commonly appeared conventional usages, we further look into some of the uncommon conventional usages. Our manual observation confirms that those uncommon conventional usages are indeed legitimate HSOs that do not appear frequently in Android apps. We present two concrete examples of uncommon conventional usages to illustrate this concept. One example is that an app first checks if the directory of downloads exists (i.e., a standard directory to place files that have been downloaded by the user), and then automatically starts the download using the *android.app.DownloadManager#enqueue* API once the download manager is ready and connectivity is available. We consider it a conventional usage because it is against the second principle of suspicious HSO's definition: the user does not intend to hide such behavior. In addition, given that there exist several substitute ways of downloading files (e.g., Http request, URLConnection, BufferedInputStream, FileOutputStream, etc.), the native APIs lie in *android.app.DownloadManager* are not that commonly used by app developers. Thus, we regarded it as an uncommon conventional usage. As another example, the sensitive behavior of vibration could be triggered only when a user clicks a certain button. We consider it also a conventional usage because it involves non-hidden behaviors. In fact, if app developers intend to hide sensitive behaviors, it would be obvious that they won't use vibration functionality to notify users. Moreover, the usage of vibration is less common because it would annoy Android users, leading to a poor user experience. Such cases appear less than 50 times in our dataset and thus we regard them as uncommon conventional usage as well.

4.5 Suspicious HSO Analysis

After eliminating conventional usages, all the remaining ones will be reported as suspicious HSOs. Among the 20,000 apps considered in this work, 1,304 of them, including 982 malware and 322 benign samples, were retained. These apps have been reported to contain in total 2,201 suspicious HSOs, with 1,790 and 441 from malware and benign apps, re-

spectively. These numbers are recapped in Table 4.1. This experimental result shows that suspicious HSOs are widely present in real-world Android apps. Figure 4.4 further illustrates the distribution of suspicious HSOs in our dataset. On average, there are 2.0 and 1.4 HSOs in each malware sample and benign app, respectively.

Table 4.1: Number of suspicious HSOs.

Initial Dataset	# HSOs	# Suspicious HSOs
10,000 benign apps	9,368 (in 3,071 apps)	441 (in 322 apps)
10,000 malicious apps	35,974 (in 5,036 apps)	1,790 (in 982 apps)
Total	45,342 (in 8,107 apps)	2,201 (in 1,304 apps)

In this work, the elimination of conventional usages is based on a pre-defined whitelist, which only includes recurrently presented HSOs in benign apps. Some less frequent yet still legitimated HSOs could have been overlooked and hence result in suspicious ones. Indeed, the remaining suspicious HSOs may not always be true positives (i.e., may contain a small number of false positives). To this end, we go one step further to calculate the precision of our approach in pinpointing suspicious HSOs in Android apps. Unfortunately, there is no known ground truth available for evaluating HSO usage in Android apps. Thus, we resort to a manual process to calculate the precision. In this work, we manually inspected the bytecode of apps to see if HiSenDroid correctly and precisely identified the suspicious trigger rather than those commonly appeared code blocks for normal usage. Here, we identify truly suspicious behavior (i.e., confirmed to be true positive) only when the HSO is security-relevant and potentially brings harm to Android users. Specifically, we rely on two principles to identify truly suspicious HSOs: (1) the hidden behavior involves security-relevant APIs that are protected by Android permissions, classified following the latest Android API-permission mappings (cf. Section 3.1), and (2) the sensitive APIs are intentionally hidden under dedicated trigger conditions. As a result, we count those who meet the two aforementioned principles as true positives. For example, if an app first intends to retrieve Device ID, and when unsuccessful, tries to read the MAC address, we will

consider it as a false positive because it is against the second principle: does not intend to hide such behavior. As another example, an app checks the build's fingerprint to see if it is running on popular emulators, and the sensitive behavior of retrieving subscriberId would be triggered only when it is not running in an emulator. We consider it as a true positive because it involves security-relevant APIs and there is sensitive behavior that is clearly hidden under trigger conditions. In our dataset, 1,304 apps have been reported to contain at least one HSO. Among the 1,304 apps, HiSenDroid has identified 14,394 HSOs, for which 2,231 of them are regarded as suspicious HSOs. By manually looking at each of those reported suspicious HSOs, we are able to confirm that 1,938 out of 2,231 of them are true positive results (or 293 of them cannot be confirmed without deeply examining the code), giving a precision of 86.8%.

Recall that the conventional usages are excluded in this work through a whitelist built through empirical evidence, and the whitelist is only considered as a configuration option to our approach. We believe that the performance of detecting suspicious HSOs could be further improved if we are able to construct a better whitelist of legitimate HSOs. This is nevertheless outside the scope of this work. We hence consider it as our future work.

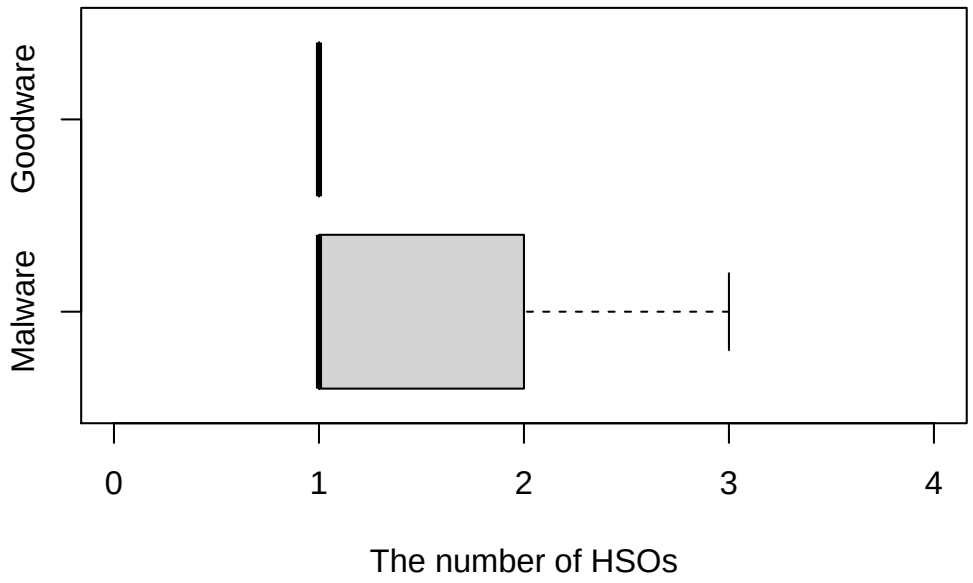


Figure 4.4: Distribution of the number of *suspicious* HSOs in *benign set* and *malware set*.

4.5.1 Trigger Conditions

Known trigger types such as time-bomb and anti-emulator techniques have been broadly studied, specific algorithms for detecting such known trigger types have been developed [Balzarotti et al., 2010, Kirat and Vigna, 2015, Fratantonio et al., 2016]. Nevertheless, the community still lacks the understanding of unknown trigger types. We therefore investigate the most frequent triggering conditions in the suspicious HSOs detected by HiSenDroid.

HiSenDroid has identified 168 unique APIs that have been traced as the source of the triggers in detected suspicious HSOs. To make it much clearer, we present all these system properties trigger conditions and environment parameters trigger conditions in the artefact package [8]. We then manually categorize them according to the types of objects they accessed. Table 4.2 illustrates the trigger condition categories and examples of system APIs that are frequently leveraged to fulfill the trigger conditions discovered in HSOs. The top trigger condition categories include time (e.g., at a certain time of a day), SMS (e.g., when receives SMS of certain formats), location (e.g., if the device is in certain countries), system property (e.g., checks the device's manufacturer), and package manager (e.g., if specific apps are installed).

Here we elaborate on each trigger condition category with real-world suspicious HSO cases identified in our dataset.

Time Triggers compare time-related properties (such as current system time, time zone, etc.) with hard-coded values to determine whether or not to execute the hidden sensitive behaviors. Listing 4.3 demonstrates a code snippet from app *com.wukongtv.wukongtv*[9], which leverages time-related triggers to hide suspicious behaviors. When the first time the app launches, it writes the timestamp into the SharedPreferences (i.e., *var0*). It then

[8] https://bitbucket.org/se_anonymous/hisendroid/src/master/experiments_results/

[9] SHA-256:3397079daa388bdbcdcc42b6834d3c792bf5c80ad24491e3893de7cfc2b11db7

Table 4.2: Categories of Trigger Conditions in HSO

Category	Most Frequent Trigger API Examples
Time	util.Calendar#get util.Date#getTime util.Calendar#getTimeInMillis
System Properties	os.Build#MODEL telephony.TelephonyManager#getSubscriberId telephony.TelephonyManager#getDeviceId
Location	telephony.TelephonyManager#getSimCountryIso telephony.TelephonyManager#getCellLocation location.LocationManager#getLastKnownLocation
SMS Message	telephony.SmsManager#divideMessage telephony.SmsManager#getDefault telephony.SmsManager#getData
Package Manager	content.Context#getPackageManager content.pm.PackageManager#getPackageInfo content.pm.PackageManager#getApplicationInfo
Miscellaneous	android.widget.CheckBox#isChecked android.app.KeyguardManager#isKeyguardLocked java.net.NetworkInterface#getHardwareAddress

compares the current system time with the first launch time (line 6); if the time interval is greater than one day, it triggers the sensitive method *bq.f()* (line 7) that retrieves the information (e.g., package name and process name) of running tasks (lines 10~15). Doing so conceals the suspicious behaviors from automatic dynamic detection, which usually starts testing immediately after the app is installed.

```
1 static void c(Context var0) {
2  Calendar var10000 = Calendar.getInstance();
3  int var2 = var10000.get(6) * 100;  //day_of_year
4  var2 += var10000.get(11);          //hours_of_day
5  // var0 is retrieved from SharedPreferences
6  if (Math.abs(var0/100L - (long) (var2/100)) >= 1L) {
7   ab.h = bq.f(var0);
8 }}}
9
10 public static Long[][] f(Context var0) {
11  var31 = var3.getRecentTasks(10, 1);
12  while(var31.iterator().hasNext()){
13   // get package name and process name of recent tasks
14   ...
15 }}
```

Listing 4.3: Code Example of Time Trigger.

System Property Triggers leverage system properties, such as the phone model, the phone number, and hardware information, to limit the sensitive behaviors within specific device brands (e.g., Samsung) or types (e.g., real device). These triggers are also commonly adopted by anti-emulator techniques to detect the presence of emulators. Listing 4.4

demonstrates an anti-emulator example extracted from app *com.gwsoft.imusic.controller* [10], which checks if the build's fingerprint contains specific strings that indicate popular emulators (line 2). The sensitive behavior of retrieving *subscriberId* (line 5) is only executed if it does not run in an emulator.

```
1 private static boolean a(Context var0) {
2  if (Build.FINGERPRINT.contains("vbox86p/vbox86p")&&
      !Build.FINGERPRINT.contains("ttVM_Hdragon/ttVM_Hdragon")&&
      !Build.FINGERPRINT.contains("generic/sdk/generic")&&
      !Build.FINGERPRINT.contains("generic_x86/sdk_x86/generic_x86")
3  // omit other strings that fingerprints popular Android emulators
4  ){
5   var2 = ((TelephonyManager)var0.getSystemService("phone")) .getSubscriberId();
6   var11.put("imsi", var2);}}
```

Listing 4.4: Code Example of System Property Trigger.

SMS Triggers Utilize the content, type, and phone number of received SMS messages to hide sensitive behaviors. An example derived from app *com.fingersoft.hillmotor*[11] is shown in Listing 4.5. When an SMS message is received, it checks the originating address of the message (line 4). If it matches a pre-defined value (e.g., 10 or 11 etc in this example), the behavior that repeatedly sends a message (line 6) to the same number via a text message service.

```
1 //var2 is the originating address retrieved from SMS
2 //var3 is the message body
3 public boolean repeat(Context var1, String var2, String var3) {
4 if ((var2.startsWith("10") || var2.startsWith("11") || var2.startsWith("12")) &&
      !var2.equals("114") && !var2.equals("12306") && !var2.equals("116114") &&
      !var2.equals("12580")) {
5 SmsManager var13 = SmsManager.getDefault();
6 var25.sendTextMessage(var2, (String)null, "Y", var16, var12);
7 }}
```

Listing 4.5: Code Example of SMS Trigger.

Location Triggers obscure sensitive behaviors with fine grained (e.g., latitude and longitude) and coarse grained (e.g. country) location information. Listing 4.6 shows an example derived from *com.inter.*

apps.patqut.apk [12] which queries the country code of the device (saved as *var1*), and checks if it is in Malaysia (line 4). If so, the app then triggers the *postLoginData2()* method (line

[10] SHA-256:8c679a7c57a7fbb355fb363d3784cc8380655701d482837869edd95f3a3ea470
[11] SHA-256:95e1cf498dec79351a9d104f5e9fb0110c267e9eff0099ada475d8832a2afb7302521
[12] SHA-256:22c9d7738073a7ac8f9b58029057c2741e89faac76b623837db2f3a8bb2d93c5

5), which retrieves the device's id (line 9) and hands it over to another activity for further malicious behaviors.

```
1 TelephonyManager var3 = (TelephonyManager)this.getSystemService("phone");
2 String var1 = var3.getSimCountryIso().toUpperCase();
3 public void getin(String var1) {
4  if (var1.equals("MY")) {
5   this.postLoginData2();
6 }}
7
8 public void postLoginData2() {
9  String var2 = ((TelephonyManager)this.getSystemService("phone")) .getDeviceId();
10 // hand over the obtained DeviceId to a new activity
11 ...
12 }
```

Listing 4.6: Code Example of Location Trigger.

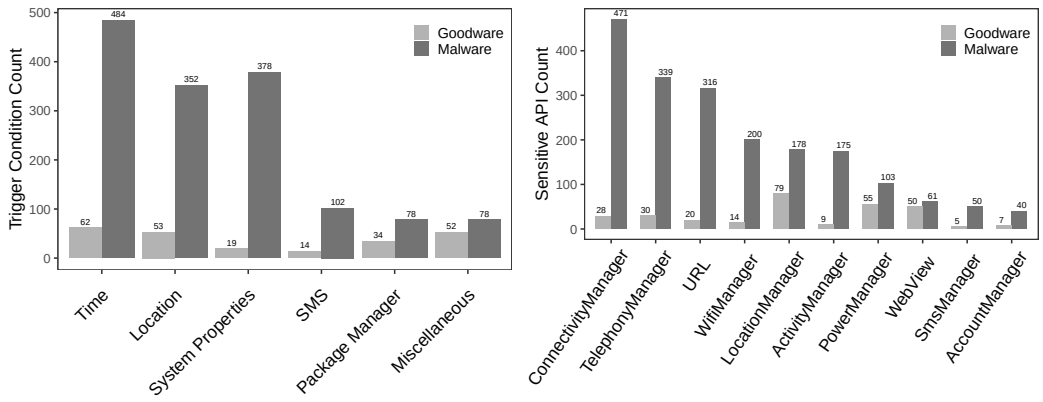
Package Manager Triggers scan the list of installed apps and inspect if specific apps (usually anti-virus tools) are installed before conducting any sensitive behaviors. Listing 4.7 shows a code snippet taken from *flash15.1.apk* [13] which searches for *AhnLab V3 Mobile Plus 2.0* (i.e., an anti-virus tool) in the list of installed apps (line 1~7). If the anti-virus tool is not installed (line 10), it then starts its malicious behaviors. Specifically, it gets the package name of the current active activity (lines 11, 12), and puts it into sleep if it is a bank app. After that, it launches a new activity that contains a phishing web page to steal user's bank credentials.

```
1 private boolean judgeAV() {
2  this.pm = this.getPackageManager();
3  this.listAppcations = this.pm.getInstalledApplications(8192);
4  for(int v = 0; v < listAppcations.size(); ++v) {
5   if(listAppcations(v).name.equalsIgnoreCase("AhnLab V3 Mobile Plus 2.0")){
6    return true;}
7   return false;}
8
9 public void run() {
10 if (!AutBan.this.judgeAV()) {
11  List var2 = ((ActivityManager)AutBan.this.getSystemService
       ("activity")).getRunningTasks(1);
12  String var1 = ((RunningTaskInfo)var2.get(0)).topActivity .getPackageName();
13  // if the top activity is a bank app, it puts the activity into sleep and start a
       phising page
14  ...
15 }};
```

Listing 4.7: Code Example of Package Manager Trigger.

Other Triggers. Besides the most frequent trigger categories, we also observed some sophisticated triggers specifically designed to counter automated dynamic testing ap-

[13] SHA-256:fdaba7f032ee7ff9adf799713b25d4c2fef86ddbbe8709bf6ec021505b8f1d0d



(a) Categories of Trigger Conditions in HSO. (b) Top 10 Categories of Sensitive APIs in Malware.

Figure 4.5: Categories of Trigger Conditions in HSO and Sensitive APIs in Malware.

proaches. Listing 4.8 shows an example taken from a music player app *com.gwsoft.imusic.controller*[14]. The app hides sensitive behaviors that retrieve the device's information (lines 8~12) behind a trigger that will only be fired when an item on the song list (i.e., *mCatalogSongsList*) is clicked (line 2). The trick here is that automated dynamic testing tools running on an emulator are likely not to have any music files and, therefore, will have no items on the list to click. Hence, only legitimate users who intend to use it to play music will have the chance to trigger the sensitive behavior.

```
1 //contains at least one song in the list
2 public void onItemClick(AdapterView<?> var1, View var2, int var3, long var4) {
3   if (mCatalogSongsList != null && var3 + -1 >= 0 && var3 + -1
      < mCatalogSongsList.size()){
4     CountlyAgent.onEvent(CuttingActivity.this,  "activity_diy_do_re",
        String.valueOf(var3));
5 }}
6
7 public static void onEvent(Context var0, String var1, String var2) {
8   HashMap var3 = new HashMap;
9   var3.put("phone", getIMSI());
10  var3.put("ip", getLocalIpAddress());
11  var3.put("app_version", versionName);
12  var3.put("imei",getDeviceId());
13 }
```

Listing 4.8: Code Example of Other Trigger.

[14]SHA-256:8c679a7c57a7fbb355fb363d3784cc8380655701d482837869edd95f3a3ea470

4.5.2 Sensitive APIs involved in suspicious HSOs

While the invocation of sensitive APIs does not necessarily mean it is malicious, sensitive APIs deliberately hidden in an HSB do raise its suspicion. HiSenDroid has identified 134 unique hidden sensitive APIs that appeared 3,195 times in our dataset. Figure 4.5(b) presents the top ten classes of the most frequently invoked sensitive APIs in *malware set* and *benign set*. The most involved APIs are network related, including the ones in *URL*, *ConnectivityManager*, and *WifiManager* classes. The*WebView* ((displays web pages)) and *SmsManager* (manages SMS operations such as sending text messages) are also prevalently used in HSOs. Other commonly involved API classes include *PowerManager* (controls the power state of the device such as keeping the screen stay on), *LocationManager* (provides access to the system location services such as getting last known location), *ActivityManager* (gives information about activities and services such as getting running tasks on the phone), *TelephonyManager* (provides access to information of the telephony services such as phone number), and *AccountManager* (manages user's online accounts). The detailed most common APIs in HSOs can be found in Table 4.3.

Interestingly, while most of the API classes have significantly more instances in malware samples than benign apps, *WebView* is an exception. We therefore took an in-depth look into benign apps with WebView APIs in their HSOs and observed that 34 out of 50 cases are free apps that display advertisement web pages for revenue.

4.5.3 Trigger Condition to Hidden Sensitive API Pairs

We now investigate the relationships between trigger conditions and the hidden sensitive APIs accessed in their corresponding HSOs so as to identify common patterns leveraged by attackers to achieve malicious purposes. Figure 4.6 graphically summarizes such relationships, i.e., trigger-to-hidden-sensitive-API pairs, where each node represents an API in either the trigger conditions or the hidden sensitive branches, while each edge denotes

Table 4.3: Details of The Top 10 Classes of Hidden Sensitive APIs in HSO

Class	Most Frequent Sensitive API Examples
ConnectivityManager	net.ConnectivityManager#getActiveNetworkInfo net.ConnectivityManager#getNetworkInfo net.ConnectivityManager#getAllNetworkInfo
TelephonyManager	telephony.TelephonyManager#getDeviceId telephony.TelephonyManager#getSubscriberId telephony.TelephonyManager#getCellLocation
URL	net.URL#openConnection net.URL#getContent net.URL#openStream
WifiManager	net.wifi.WifiManager#getScanResults net.wifi.WifiManager#getConnectionInfo net.wifi.WifiManager#getWifiState
LocationManager	location.LocationManager#getLastKnownLocation location.LocationManager#requestLocationUpdates location.LocationManager#getBestProvider
ActivityManager	app.ActivityManager#getRunningTasks app.ActivityManager#getRecentTasks app.ActivityManager#moveTaskToFront
PowerManager	os.PowerManager.WakeLock#release os.PowerManager.WakeLock#acquire() os.PowerManager.WakeLock#acquire(long)
WebView	webkit.WebView#setBackgroundColor webkit.WebView#addJavascriptInterface webkit.WebView#loadDataWithBaseURL
SmsManager	telephony.SmsManager#sendTextMessage telephony.SmsManager#sendMultipartTextMessage telephony.SmsManager#sendDataMessage
AccountManager	accounts.AccountManager#getAccountsByType accounts.AccountManager#getAccounts accounts.AccountManager#getUserData

the connections between them. HiSenDroid has identified 404 nodes within which 346 are APIs in trigger conditions, 134 are APIs in hidden sensitive branches, and 15 APIs exist in both triggers and hidden sensitive branches. There are 2,847 edges found between them, which are illustrated in different colors according to their trigger conditions' categories.

Table 4.4 further details the top 10 pairs found in HSOs with their categories and counts. The most frequent HSO patterns are to hide network-related activities behind retrieving the phone's location. For instance, the top one pattern that appeared 135 times in our dataset requests the SIM provider's country code. Based on the user's location, it then determines whether or not to open a web page, and what web pages (e.g., advertisement pages) to display to the user. Time-related HSO patterns are also widely found in the detected HSOs. They firstly compare the current system time with preset values. If the

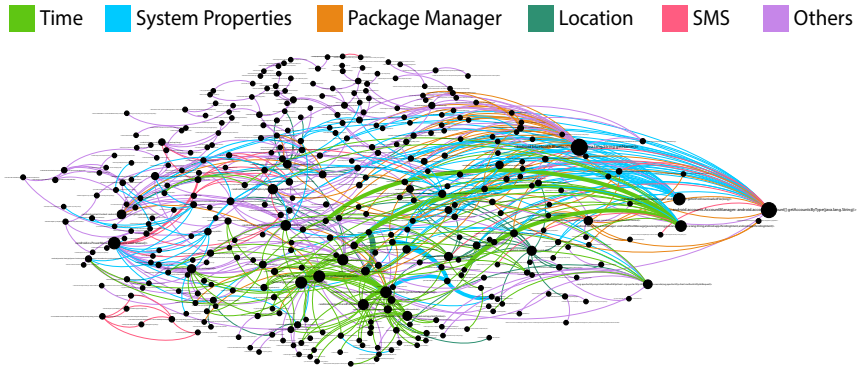


Figure 4.6: The HSO Trigger-Sensitive API Pairs Graph.

condition fulfills (e.g., the app is running for more than ten minutes), they try to initialize a network connection and send out the user's private information such as IMEI, phone number, etc. More than 250 instances in our dataset leverage this pattern to steal users' private information stealthily. Other frequent HSO patterns on the top list are involved in anti-emulator tricks include checking the phone's model name and checking if specific apps are installed (which could indicate if it is an emulator) before acquiring sensitive information.

Table 4.4: Top 10 Trigger Condition to Hidden Sensitive API Pairs.

Category	Trigger Condition APIs	Hidden Sensitive APIs	Counts
Location	TelephonyManager#getSimCountryIso	java.net.URL#openConnection	135
Location	TelephonyManager#getCellLocation	ConnectivityManager#getActiveNetworkInfo	131
Time	java.lang.System#currentTimeMillis	ConnectivityManager#getNetworkInfo	80
Time	java.lang.System#currentTimeMillis	ConnectivityManager#getAllNetworkInfo	66
Time	java.lang.System#currentTimeMillis	ConnectivityManager#getActiveNetworkInfo	57
System Properties	android.os.Build#MODEL	TelephonyManager#getSubscriberId	54
System Properties	android.os.Build#MODEL	Settings$System#putInt	54
SMS	android.os.Message#obj	PowerManager$WakeLock#release	52
Package Manager	PackageManager#getInstalledPackages	ActivityManager#getRunningTasks	51
Time	java.lang.System#currentTimeMillis	DefaultHttpClient#execute	49

4.5.4 Suspicious HSOs in Third-party Code

Finally, we further look into the identified HSOs to check if they are introduced by app developers or reused by third-party libraries. For the sake of simplicity, we consider the code only located in the unique app package (also known as the app id) as developers newly

implemented code while all the other code (i.e., in packages not connected with the app's domain name) as third-party code (e.g., third-party libraries). Among the 2,201 HSOs, surprisingly, over half of them (i.e., 1,342) is contributed by third-party code (i.e., 1,173 HSOs in malware and 169 in benign apps), among which malware tends to be more favored to introduce HSOs through third-party code than benign apps. This experimental evidence suggests that attackers have more incentives to achieve malicious behaviors through third-party code as it allows easy code reuse that makes it much easier to implement new malware.

4.5.5 Comparison with state-of-the-art

We now compare our approach with state-of-the-art works targeting the problem of detecting hidden sensitive operations. To the best of our knowledge, there are two closely related approaches: HSOMiner[Pan et al., 2017] and TriggerScope[Fratantonio et al., 2016]. Unfortunately, the source code of HSOMiner is not publicly available, and it is infeasible to compare against it because they trained data on the authors' labelled dataset, which has also not been publicly released. We have contacted the authors about launching their approach to analyze Android apps. Unfortunately, we have not yet received any response from them. Similarly, the authors of TriggerScope have also not made it publicly available. As a result, we cannot compare with TriggerScope as well. Fortunately, Jordan Samhi has provided a re-implemented version[15] of TriggerScope based on the details given in its research paper. The re-implemented version is named as TSOpen (referring to as the open implementation of TriggerScope) and has already been leveraged by previous studies [Samhi and Bartel, 2021]. In this work, we resort to comparing our approach with TriggerScope by actually comparing it with TSOpen.

To set up the experiments for a fair comparison, we run TSOpen on the same 10,000 malware and 10,000 benign apps selected in evaluating HiSenDroid in section 4 (as indi-

[15]https://github.com/JordanSamhi/TSOpen

cated in the second and third columns in Table 4.5). The experiments are executed under the same environment, i.e., the same server and the same timeout threshold (i.e., 20 minutes).

The experimental results are summarized in Table 4.5. Overall, the number of HSOs found by HiSenDroid in goodware and malware (i.e., 441 and 1,790, respectively) is larger than those found by the TSOpen (i.e., 110 and 237, respectively). Recall that when evaluating the performance of HiSenDroid at the beginning of Section 4.5, we have manually validated the 2,231 suspicious HSOs yielded by HiSenDroid, for which 1,938 are confirmed to be true positives, giving a precision of 86.8%. In this work, we further conduct the same manual validation for the results of TSOpen. Our manual validation confirms that TSOpen has at least correctly detected 90.2% of logic bombs. This result is expected as TSOpen only detects three types of HSOs (i.e., time, location, and SMS) while HiSenDroid aims at detecting a broader scope of HSOs. To enable a fair comparison[16], in this work, we will only consider HiSenDroid's results falling in these three categories.

As highlighted in Table 4.5 (cf. Columns 6-8), HiSenDroid detects more HSOs in all of the three categories. Among the detected HSOs, we find that 265 HSOs (186, 48, and 31 in time, location, and SMS, respectively) were detected by both tools (as summarized in the fourth row in Table 4.5). Besides that, there are 802 HSOs (360, 357 and 85 in time, location, and SMS, respectively) exclusively detected by HiSenDroid, while still 82 HSOs (38, 1, and 43 in time, location, and SMS, respectively) identified by TSOpen are not flagged by HiSenDroid.

On a further investigation, we found the reason why HiSenDroid failed in detecting the 82 HSOs is that HiSenDroid's definition of potentially-sensitive APIs is different from the definition in TSOpen. In this work, we consider all the APIs that are protected by Android permissions as potentially sensitive, while TSOpen takes a different approach to pre-select

[16]We consider the original outputs of HiSenDroid and TSOpen for comparison since only a small number of their results could be false positive.

Table 4.5: The comparison results between HiSenDroid and TSOpen.

Tool Name	# Analyzed Goodware	# Analyzed Malware	# HSOs in Goodware	# HSOs in Malware	# Time HSOs	# Location HSOs	# SMS HSOs
HiSenDroid	10,000	10,000	441(in 322 apps)	1,790(in 982 apps)	546	405	116
TSOpen	10,000	10,000	110(in 51 apps)	237(in 123 apps)	229	49	69
Common	10,000	10,000	71	194	186	48	31

such a set of sensitive APIs[17]. Their set of sensitive APIs includes both permission-protected and permission-free APIs. For example, TSOpen treats the following two APIs, <android.content.BroadcastReceiver: void abortBroadcast()> and <android.os.Handler: boolean sendEmptyMessage(int)>, as sensitive APIs. However, HiSenDroid does not consider them as sensitive because they are not protected by permissions. Furthermore, considering that TriggerScope was published in 2016 and the Android API rapidly evolves, it is understandable that certain APIs (especially the latest ones) are not included, resulting in possibly less suspicious HSOs. Moreover, as claimed in their paper, TriggerScope only focused on characterizing logic bombs on some given behaviors, while HiSenDroid treated each sensitive API in state-of-the-art Android API-permission mappings [Au et al., 2012, Backes et al., 2016, Aafer et al., 2018, Li et al., 2022a] as a target API, leading to better performance in terms of both quantity and variety in detected HSOs, compared with TriggerScope.

4.5.6 Impact of Code Obfuscation

As experimentally revealed by Zeng [Zeng et al., 2018] and Moser et al. [Moser et al., 2007], trigger conditions of HSOs could be obfuscated in order to evade the detection of advanced semantics-based malware analyzers. Therefore, we are interested in checking to what extent our approach is impacted by obfuscation, especially when applied to pinpoint HSOs in real-world Android apps. Since there is no existing dataset that is suitable for our experiment, we resort to preparing such a dataset from scratch, i.e., to form a set of obfuscated app pairs for which each pair contains a non-obfuscated app and its obfuscated

[17]The sensitive APIs are a part of internal implementation of TSOpen, which is not configurable.

counterpart. We start by randomly selecting 1,000 malware from our dataset and then apply Obfuscapk[Aonzo et al., 2020] on them to generate their obfuscated counterparts. Obfuscapk is a modular Python tool designed to directly obfuscate closed-source Android apps. Obfuscapk supports six types of obfuscation operations, which could be configured to achieve different granularities when obfuscating Android apps.

The six types of operations are summarized as follows.

1. Nop: Insert junk code. Nop, short for no-operation, is a dedicated instruction that does nothing. This technique just inserts random nop instructions within every method implementation.

2. Rename: operations that change the names of the used identifiers (classes, fields, methods).

3. Reorder: This technique consists of changing the order of basic blocks in the code. When a branch instruction is found, the condition is inverted (e.g., branch if lower than, becomes branch if greater or equal than) and the target basic blocks are re-ordered accordingly. Furthermore, it also randomly rearranges the code abusing goto instructions.

4. Reflection: This technique analyzes the existing code looking for method invocations of the app, ignoring the calls to the Android framework (see AdvancedReflection). If it finds an instruction with a suitable method invocation (i.e., no constructor methods, public visibility, enough free registers etc.) such invocation is redirected to a custom method that will invoke the original method using the Reflection APIs.

5. Advanced Reflection: Uses reflection to invoke dangerous APIs of the Android Framework. To find out if a method belongs to the Android Framework, Obfuscapk refers to the mapping discovered by Backes et al. [Backes et al., 2016]

6. Encryption: packaging encrypted code/resources and decrypting them during the app execution. When Obfuscapk starts, it automatically generates a random secret key (32 characters long, using ASCII letters and digits) that will be used for encryption.

In this work, we are interested in checking the impact of all of these six types of operations on our approach. Hence, for each of the selected apps and each obfuscation type, we launch Obfuscapk to generate an obfuscated app. For the 1,000 selected apps, we expect to generate 6,000 obfuscated apps and eventually form 6,000 obfuscated app pairs. We then launch HiSenDroid to analyze those apps and compare the number of detected HSOs obtained on apps with and without obfuscation. Table 4.6 summarizes our experimental results.

Expectedly, except for reflection, our approach is resilient to all the other four obfuscation types. Our deep analysis reveals that the reason why HiSenDroid is unaffected by Nop obfuscator is that Nop obfuscator will only insert junk code, which is a dedicated instruction that does nothing. In terms of Rename and Reorder obfuscator, their code transformations will retain the functionality as the original APK thus will not impact our approach. Also, the reason why the Encryption obfuscator has no effect on HiSenDroid is that it will only encrypt constant strings in code, which will not impact the data flow analysis of our approach. In terms of reflection obfuscator and advanced Reflection obfuscator, both trigger conditions and sensitive API invocations can be redirected to other code entities by reflection calls, while those entities cannot be always resolved statically since the reflection call targets may not be statically resolved, which would lead to false negatives of HiSenDroid. The remaining two types that have an impact on our approach are all related to reflection, which performs complicated code changes that will likely break the data flow processes. Nevertheless, even for reflection, our approach can still detect around one-third of HSOs.

To better mitigate the impact of reflection-based obfuscation on our approach, we fur-

Table 4.6: The comparison results of HiSenDroid before and after obfuscation techniques in malware.

Obfuscator	Nop	Rename	Reorder	Reflection	Advanced Reflection	Encryption
# HSOs Before Obfuscation	144(in 821 apps)	18(in 332 apps)	146(in 792 apps)	13(in 345 apps)	145(in 821 apps)	141(in 811 apps)
# HSOs After Obfuscation	144(in 821 apps)	18(in 332 apps)	146(in 792 apps)	4(in 345 apps)	64(in 821 apps)	141(in 811 apps)
Common	144	18	146	4	64	141

ther propose to strengthen the capability of HiSenDroid by integrating the state-of-the-art reflection analysis tool DroidRA to handle reflection usages [Sun et al., 2021b]. After statically locating the reflective calls, DroidRA can transform a reflection-included Android app to a reflection-free version, where the located reflective calls will be represented by standard java calls. The newly generated reflection-free app would allow HiSenDroid to yield reflection-aware analysis results. Specifically, considering the 345 apps and 821 apps that are obfuscated by reflection calls and advanced reflection calls, respectively, we first apply DroidRA to convert them into 1,166 reflection-free apps. After that, we execute HiSenDroid to perform HSO analysis on these new apps and compare the number of detected HSOs obtained based on the original apps. As a result, HiSenDroid is able to detect all 13 reflection-relevant HSOs which are obfuscated with reflection obfuscation, while detecting 124 (with a success rate of 85.5%) reflection-relevant HSOs that are obfuscated with advanced reflection obfuscation. The reason why HiSenDroid fails on detecting a small portion of reflection-relevant HSOs is that DroidRA may not resolve all the advanced reflective calls. For example, DroidRA relies on COAL [Octeau et al., 2015] solver to infer reflective calls, which might introduce false negatives, leading to reflection calls unresolved and thus can not be successfully detected by HiSenDroid. Nevertheless, our experimental result shows the capability of HiSenDroid in achieving most of the reflection-aware hidden sensitive operation detections.

4.6 Implication

After being able to automatically detect suspicious HSOs, we now go one step further to investigate how such HSOs can bring security harms to users. There might be different

security implications, in this work, we only focus on sensitive data leaks, which is also part of our initial attempts towards demonstrating the usefulness of identifying suspicious HSOs. Specifically, we are interested in detecting hidden sensitive data flows (HSDFs), i.e., leaking sensitive data collected through HSOs. To the best of our knowledge, hidden sensitive data flow has not yet been explored by our community. Unfortunately, it has not even been clearly defined. To this end, we first define HSDF following the previous rules leveraged to define HSOs (cf. Section 4.2). Let S denote a sensitive data flow (also known as a private data leak as mentioned in the FlowDroid work [Arzt et al., 2014]), we consider that a sensitive data flow happens when a sensitive "tainted" information goes from a source (e.g. the API method *getDeviceId*) to a given sink (e.g. the API method *sendTextMessage*).

Definition 3 [Hidden Sensitive Data Flow (HSDF)]: A sensitive data flow S is an HSDF if the source of S appears in the hidden sensitive branch of a HSO.

Although HSDFs have not yet been specifically exploited by the state-of-the-art, our community has proposed various approaches to detect general sensitive data-flows. One of the most famous approaches is FlowDroid [Arzt et al., 2014], a state-of-the-art static analyzer that performs taint analysis to pinpoint sensitive data leaks flowing from a pre-defined set of *source* methods to *sink* methods. These *source* and *sink* methods can be easily customized. In this work, we leverage FlowDroid to detect sensitive data flows related to HSOs. If a sensitive data flow reported by FlowDroid has its source method invoked in an HSO, we regard it as an HSDF.

By applying FlowDroid[18] to 1,304 apps (982 malware and 322 goodware) involving suspicious HSOs, we find that 67 apps further involve HSDFs, accounting to in total 401 HSDFs. While manually checking the experimental results of FlowDroid and HiSenDroid, we find that 16 sensitive APIs, which are frequently invoked within HSOs to collect system information, are not taken into account by the source set of FlowDroid by default. These

[18]In this work, the latest *development* branch of FlowDroid[Flo,] is leveraged for the experiments. It should be roughly equivalent to the FlowDroid 2.8 release.

APIs (listed in Table 4.7), after manual confirmation, should still be considered as source methods by FlowDroid as they are responsible for retrieving sensitive data that should not be exposed to other parties. Here, to clarify, when doing the experiment, we include both of the default source and sink methods of FlowDroid and the additional sensitive APIs involved in HSOs in the SourceAndSink.txt file of FlowDroid. During the manual process, we have not found any sensitive API (i.e., involving dangerous operations) that should be additionally considered as a *sink* method by FlowDroid. Hence, we add the 16 APIs to the source set of Flowdroid and keep its sink set unchanged (hereinafter referred to this version as FlowDroid + HiSenDroid) and relaunch it on the same set of apps. This time, we are able to disclose 1,110 HSDFs from 1,304 apps. This result shows that suspicious HSOs could be leveraged to leak users' sensitive information outside of their devices. As an example shown in Listing 4.5, the sensitive data *device id* and *subscriber id*, which are unique to the device and hence can be leveraged to uniquely track the phone, are eventually sent outside the device through a text message.

Considering general sensitive data-flows (SDF), we compare FlowDroid with HiSenDroid on the same dataset. In general, among the 1,304 apps, HisenDroid+FlowDroid detect 31,215 SDF, which is significantly larger than that of the original FlowDroid (which is 16,946). This result, as expected[19], does experimentally demonstrate the effectiveness of our approach towards revealing more data flows in Android Apps. Our experimental results are illustrated in Figure 4.7, which indicates the distribution of the number of sensitive data flows in each app yielded by HisenDroid+FlowDroid and HiSenDroid. This result shows that FlowDroid + HiSenDroid has significantly improved the original results of FlowDroid, which shows the usefulness of our identified HSOs and suggests that there is a strong need to characterize hidden sensitive operations.

[19]We remind the readers that, in this work, we did not improve FlowDroid by itself but only enlarged its *source* set as some of the sensitive APIs, which are favored by HSOs, are overlooked by FlowDroid.

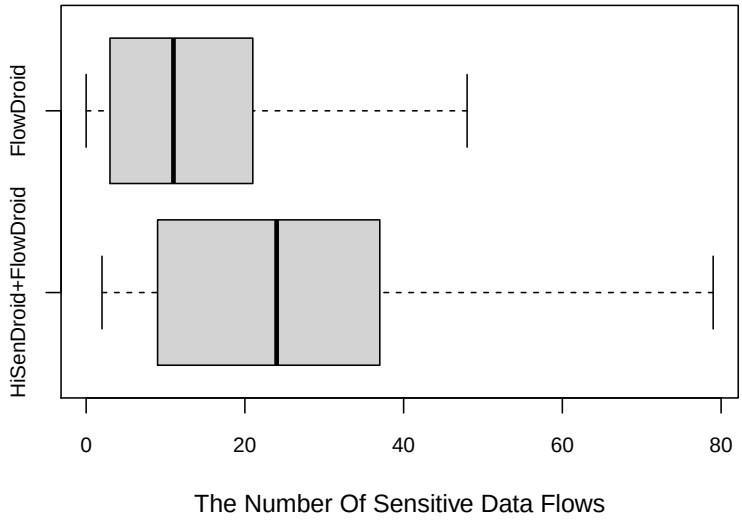


Figure 4.7: Results of Sensitive data Flows in Android Apps.

Table 4.7: The list of selected source methods that, by default, are not included by FlowDroid.

API Signature
android.net.wifi.WifiManager#getConnectionInfo()
android.app.ActivityManager#getRunningTasks
android.app.ActivityManager#getRecentTasks
android.accounts.AccountManager#getUserData
android.net.ConnectivityManager#getNetworkInfo
android.provider.Settings$System#getUriFor
android.telephony.TelephonyManager#getNeighboringCellInfo
android.telephony.TelephonyManager#getCellLocation
android.accounts.AccountManager#getAccountsByType
android.net.wifi.WifiManager#getScanResults
android.net.wifi.WifiManager#getConfiguredNetworks
java.net.URL#openConnection
android.net.ConnectivityManager#getAllNetworkInfo
android.net.VpnService#prepare
android.hardware.Camera#open
android.net.ConnectivityManager#getActiveNetworkInfo

4.7 Limitations

The main limitation of our approach lies in the backward data-flow analysis, which applies only context-insensitive analysis and thereby may lead to imprecise results. Furthermore, at the moment, our approach is not aware of dynamically loaded code, reflectively accessed methods, and native code. Subsequently, HiSenDroid may overlook certain app features and hence result in false-negative results.

Second, HiSenDroid data-flow analysis may be susceptible to obfuscation techniques.

According to some former research works [Glanz et al., 2020, Rasthofer et al., 2016, Samhi et al., 2022], obfuscation (especially those involving complicated changes of the program code) may cause false negatives of the static analysis approach. Indeed, as demonstrated by Moser [Moser et al., 2007], obfuscation is actually a challenge for almost all static program analyzers. Just like all prior efforts on static analysis of HSOs [Fratantonio et al., 2016], [Pan et al., 2017], we do not consider the apps whose branch conditions have been deeply obfuscated. Fortunately, the majority of obfuscations applied to Android apps only involve basic transformations (such as renaming [Dong et al., 2018c]) that do not involve complicated code changes (e.g., structural or logic changes, or invoke sensitive code through reflections, etc.), which will not impact the analysis of our approach. This has also been confirmed by our exploratory study towards understanding the impact of obfuscation on our approach, as discussed in Section 4.5.6. Considering reflection obfuscation, integrating DroidRA with HiSenDroid as a pipeline is demonstrated to be effective in eliminating the impact of reflection calls. Therefore, we believe that the technical capabilities and our results would not be significantly impacted by code obfuscation. Nevertheless, as part of our future work, we plan to integrate other approaches developed by our fellow researchers to mitigate these long-standing challenges, e.g., by applying DroidRA [Sun et al., 2021b, Li et al., 2016b] to mitigate the impact of reflection-enhanced code obfuscations.

Although summarized from many sensitive operations, the definition of HSO rules may not be perfect. Indeed, on the one hand, the set of sensitive operations considered for summarization may not be representative, and the set of apps leveraged to obtain such sensitive operations may not be represented as well. On the other hand, the manual analysis leveraged to summarize the rules may contain errors since it is known that human efforts are prone to errors. Apart from that, the definition of HSO is based on empirical evidence that might not be perfect. There might be complicated cases that do not follow the definition but still manifest themselves as hidden sensitive behaviors in practice, leading to false

negatives. This limitation can also apply to the conventional usage analyses since the list of conventional usages is manually summarized based on a given set of apps. The subsequent outputs (i.e., whitelist) may not be representative. Nonetheless, our follow-up study using a set of 20,000 new apps has shown that this impact is negligible. Furthermore, in this work, we have attempted to provide detailed insights to explain why HSOs are reported as such. This knowledge is expected to be useful for practitioners and researchers to characterize conventional usages and for security analysts to understand suspicious HSOs.

Moreover, since the original implementation of TriggerScope is not publicly available, we have resorted to an open re-implementation version of TriggerScope to compare our approach against it. This alternative decision may result in possible biases as the re-implementation may not really represent the original version. Unfortunately, the re-implemented version is the only source we can publicly locate to fulfill the comparison. As of our future work, we plan to also evaluate the reliability of the re-implementation of TriggerScope so as to mitigate potential biases, if any.

Last but not the least, the performance of the hidden sensitive data flow analysis may be impacted by the collection of sensitive APIs (i.e., sources). On one hand, some sensitive APIs, especially the latest ones, might be overlooked by FlowDroid and hence cannot be considered for pinpointing potential leaks, leading to false-negative results. In this work, our experimental results have confirmed this. On the other hand, some historical sensitive APIs included in FlowDroid's source list might be deprecated and subsequently removed from a certain Android API version [Li et al., 2018b]. There is hence no need to include them when analyzing apps targeting higher API versions, as these APIs will not be used anymore, not even mentioning causing sensitive data leaks. To overcome these impacts, we believe there is a need to keep updating FlowDroid's list of sensitive APIs, in order to achieve a more effective and sound sensitive data flow analysis for Android apps. Furthermore, ideally, FlowDroid should also not be expected to include APIs that are released after

itself.

4.8 Related Work

Hidden sensitive operations have long existed in Android malware as evasive technologies have widely been used by attackers to hide their malicious behaviors. Our research community has hence proposed various approaches to tackle these issues. We now discuss some of the representative works from two angles, including the evasive techniques that have been proposed to hide malicious code from being identified, and the detection methods proposed to pinpoint such evasive techniques.

Evasive Techniques. There has been a number of research works on hiding malicious behavior from detection, most of which focus on evading the dynamic test platforms such as virtual machines and emulators. Early works target the Windows platform [Chen et al., 2008], while recently the trend has been moved to Android [Petsas et al., 2014, Vidas and Christin, 2014, Jing et al., 2014, Diao et al., 2016, Costamagna et al., 2018, Liu et al., 2021c, Liu et al., 2022c]. These evasive techniques detect the presence of a simulated environment by either looking into the system properties of the testing platform (e.g., system fingerprints, hardware capabilities, etc.) [Petsas et al., 2014, Vidas and Christin, 2014, Costamagna et al., 2018], or leveraging a reverse Turing test that examines if the app interacts with a human user [Diao et al., 2016]. For instance, Diao et al. [Diao et al., 2016] observed that programmed interaction has specific patterns of input and interaction frequency, which is different from real users. Overall, the evasive techniques usually hide malicious activities in an *if-then-else statement*. The hidden malicious behavior will only be set off when certain conditions are fulfilled (e.g., not in an emulator); otherwise dummy benign operations are triggered. The prevalence of such evasive techniques motivated us to investigate the HSOs in Android apps and propose HiSenDroid to detect them.

Detection of Evasive Techniques. The pervasive evasive techniques (e.g., anti-

emulator techniques) have motivated the research community to take countermeasures. Great effort has been spent on detecting known types of hidden behaviors that hampers the dynamic analysis process. These works include detecting anti-emulator techniques [Balzarotti et al., 2010, Lindorfer et al., 2011, Kirat et al., 2014, Kirat and Vigna, 2015] and generic logic-bombs [Crandall et al., 2006, Brumley et al., 2008, Zheng et al., 2012, Fratantonio et al., 2016, Papp et al., 2019]. The approaches of detecting anti-emulator techniques compare the behavioral deviation of the tested apps on the various environments when feeding them the same input. The fundamental idea is that if the app behaves differently in different environments, it is likely trying to evade one or more analysis platforms (usually referred to as bare-metal analysis in the literature) [Balzarotti et al., 2010, Lindorfer et al., 2011, Kirat et al., 2014, Kirat and Vigna, 2015]. While these early works investigate a critical category of hidden operations (i.e., anti-emulator), the proposed methods lack generalization that cannot be applied to detect other types of hidden operations emerging recently.

Besides the detection of anti-emulator techniques, several works are focusing on uncovering other trigger-based behaviors. These approaches leverage symbolic execution or static code analysis and instrumentation to expose the hidden branches in an *if-then-else statement* [Crandall et al., 2006, Brumley et al., 2008, Zheng et al., 2012, Fratantonio et al., 2016, Papp et al., 2019]. As examples, Zheng et al. [Zheng et al., 2012] proposed to leverage a static analysis approach to retrieve all UI related events, and use dynamic testing to trigger them and log the invocation of sensitive APIs. Unlike HiSenDroid that leverages static analysis, the dynamic analysis based approach introduces significant system- and time-overhead. The coverage of the dynamic analysis is also in question. Fratantonio et al. [Fratantonio et al., 2016] proposed TriggerScope to detect hidden triggered behaviors based on the observation that certain triggers (i.e., time, location, and SMS related triggers) always involve the comparison of specific types of input (i.e., system time, system

location, and received SMS). Symbolic execution is then leveraged to detect such narrow conditions. While TriggerScope is effective in detecting the above-mentioned three types of logic bombs, it cannot be generalized to detect hidden operations triggered by other types of conditions, such as system property, which has been found pervasive in Android apps.

Similar to HiSenDroid, another line of work attempts to detect unknown types of trigger-based behaviors [Pan et al., 2017], [Wang et al., 2017]. A prominent example is HSOMiner [Pan et al., 2017], which extracts static characteristics of hidden behaviors as features and trains a machine learning model to identify the code blocks that observe similar patterns. The major differences between our work and HSOMiner are twofold. First, HSOMiner requires a large number of manually labelled training samples, which involves extensive human experts' effort. Its performance also heavily relies on the manually labelled training data, which is prone to errors. Our method, on the other hand, is an automatic process without human intervention. Second, HSOMiner, as a machine learning based approach, lacks explanations of the decisions. In contrast, our static code analysis based approach outputs the full call traces of detected HSOs, and provides more detailed information for further analysis.

4.9 Summary

In this work, we present to the community a prototype tool called HiSenDroid, which performs a static code analysis to uncover hidden sensitive operations that will only be triggered under special circumstances such as at a specific location or in a certain time period. Additionally, HiSenDroid goes one step deeper to provide details aiming at helping security analysts understand why a given hidden sensitive operation is flagged as such. Experimental results over 20,000 apps, including both malicious and benign apps, show that hidden sensitive operations are indeed quite frequently presented in Android apps and HiSenDroid

is effective in automatically discovering them. Moreover, with the help of FlowDroid, a state-of-the-art static taint analyzer, we further experimentally find that hidden sensitive operations could eventually lead to privacy leaks.

Chapter 5

Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues

Xiaoyu Sun, Xiao Chen, Yanjie Zhao, Pei Liu, John Grundy and Li Li. 2022. Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues. The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2022. https://arxiv.org/abs/2208.13417

5.1 Introduction

Unit testing is a type of software testing aiming at testing the effectiveness of a software's units, such as functions or methods. It is often the first level of testing conducted by software developers themselves (hence a white box testing technique) to ensure that their code is correctly implemented. Unit testing has many advantages. First, it helps developers fix bugs early in the development cycle which subsequently saves costs in the end. Indeed, it is known that the cost of a bug increases exponentially with time in the software development workflow. Second, it makes it possible to achieve regression testing. For example, if developers refactor their code later, it allows them to make sure that the refactored code still works correctly. Third, unit tests provide an effective means for helping developers understand the unit under testing, i.e., what functions are provided by the unit and how to use them? Because of the aforementioned advantages, it is recommended to always write unit tests when developing software, and the unit tests should cover as many units as possible.

The Android framework, as one of the largest software projects (with over 500,000 commits), is no exception. The Android framework provides thousands of public APIs that

are heavily leveraged by app developers to facilitate their development of Android apps. Ideally, each such public API should be provided with a set of unit tests to ensure that the API is correctly implemented and the continuous evolution of the framework will not change its semantics. Unfortunately, based on our preliminary investigation, less than 30% of APIs, are provided with unit test cases, leaving the majority of APIs uncovered. This is unacceptable considering that the Android framework nowadays has become one of the most popular projects (with millions of devices running it).

This poor test coverage of Android APIs has led to serious compatibility issues in the Android ecosystem, as recently shown [Li et al., 2018b, Wei et al., 2016, Mutchler et al., 2016, Zhang et al., 2015, Ham and Park, 2011, Huang et al., 2018]. For example, Li et al. [Li et al., 2018b] demonstrate that various Android APIs suffer from compatibility issues as the evolution of the Android framework will regularly remove APIs from or add APIs into the framework. Such API removal or addition can result in no such class or method runtime exceptions when the corresponding app is running on certain framework versions. Liu et al. [Liu et al., 2021a] further present an approach to detect silently-evolved Android APIs, which could cause another type of compatibility issue as their semantics are altered (while not explicitly documented) due to the evolution of the Android framework. Moreover, Wei et al. [Wei et al., 2019] experimentally show that some Android APIs could even be customized by smartphone manufacturers, leading to another type of compatibility issue that causes Android apps to crash on certain devices while behaving normally on others. The authors further propose a prototype tool called PIVOT to automatically learn device-specific compatibility issues from existing Android apps. Their experiments on a set of top-ranked Google Play apps have discovered 17 device-specific compatibility issues.

To the best of our knowledge, the state-of-the-art works targeting compatibility issue detection leverage static analysis techniques to achieve their purpose. However, as known to the community, the static analysis will likely yield false positive results as it has to

make some trade-offs when handling complicated cases (e.g., object-sensitive vs. object-insensitive). In addition, the static analysis will also likely suffer from soundness issues because some complicated features (e.g., reflection, obfuscation, and hardening) are difficult to be handled [Sun et al., 2021b, Samhi et al., 2022]. Furthermore, except for syntactic changes, compatibility issues could also be triggered by semantic changes, which are non-trivial to be handled statically. Indeed, as demonstrated by Liu et al. [Liu et al., 2021a], there are various semantic change-induced compatibility issues in the Android ecosystem that remain undetected after various static compatibility issue detection approaches are proposed to the community.

Moreover, static app analysis can only be leveraged to perform post-momentum analysis (i.e., after the compatibility issues are introduced to the community). They cannot stop the problems from being distributed into the community – many Android apps, including very popular ones, still suffer from compatibility issues. To mitigate this problem, we argue that incompatible Android APIs should be addressed as early as possible, i.e., ideally, at the time when they are introduced to the framework. This could be achieved by providing unit tests for every API introduced to the framework and regressively testing the APIs against Android devices with different manufacturers and different framework versions. However, it is time-consuming to manually write and maintain unit tests for each Android API (which probably explains why there is only a small set of APIs covered by unit tests at the moment). There is hence a need to automatically generate compatibility unit tests for Android APIs.

In this work, we present a prototype tool, JUnitTestGen, that attempts to automatically generate test cases for Android APIs based on their practical usage in real-world apps. Specifically, after locating existing API usages in real-world Android apps, JUnitTestGen performs field-aware, inter-procedural backward data-flow analysis to infer the API caller instance and its parameter values. JUnitTestGen then leverages the inferred information

to reconstruct a minimal executable code snippet for the API under testing. Experimental results on thousands of Android apps show that JUnitTestGen is effective in generating test cases for Android APIs. It achieves an 80.4% of success rate in generating valid test cases. These test cases subsequently allow our approach to pinpoint various types of compatibility issues, outperforming a state-of-the-art generic test generation tool named EvoSuite, which can only generate test cases to reveal a small subset of compatibility issues. Furthermore, we demonstrate the usefulness of JUnitTestGen by comparing it against a state-of-the-art static analysis-based compatibility issue detector called CiD. JUnitTestGen is able to mitigate CiD's false-positive results and go beyond CiD's capability (i.e., detecting compatibility issues induced by APIs' signature changes) to detect compatibility issues induced by APIs' semantic changes.

Overall, we make the following main contributions in this work:

- We have designed and implemented a prototype tool JUnitTestGen that leverages a novel approach to automatically generate unit test cases for APIs based on their existing usages.
- We have set up a reusable testing framework for pinpointing API-induced compatibility issues by automatically executing a large set of unit test cases on multiple Android devices.
- We have demonstrated the effectiveness of JUnitTestGen by i) generating valid test cases for Android APIs and pinpointing problematic APIs that could induce compatibility issues if accessed by Android apps, ii) outperforming state-of-the-art tools on real-world apps in detecting a wider range of compatibility issues.

The source code[1] and experimental results are all made publicly available in our artifact package.[2].

[1] https://github.com/SMAT-Lab/JUnitTestGen
[2] https://doi.org/10.5281/zenodo.6507579

5.2 Motivation

To overcome the fragmentation problem, our fellow researchers have proposed various approaches to mitigate the usage of compatibility issues in Android apps [Wei et al., 2016, Wei et al., 2019, Xia et al., 2020]. These approaches mainly leverage static analysis to achieve their purpose. Unfortunately, static analysis is known to likely generate false-positive and false-negative results and is yet hard to handle such issues that involve semantics changes in Android APIs [Liu et al., 2021a]. Therefore, we argue that there is a need also to invent dynamic testing approaches to complement existing works in handling app compatibility issues.

We hence start by conducting a preliminary study investigating the test case coverage in the Android framework. Specifically, we downloaded the source code of AOSP from API level 21 to 30 and then calculated the number of public APIs[3] and their corresponding unit test cases provided by Google. Our result reveals that on average **less than 30% of Android framework APIs have provided test cases in each API level**, indicating the Android framework has a poor test case coverage. When more APIs are provided with unit test cases, more compatibility issues of APIs will likely be identified during regression testing. This will enable them to be fixed at an earlier stage to avoid the introduction of compatibility issues in the first place. To this end, we propose to effectively and efficiently detect compatibility issues through a dynamic testing approach that fulfills its objective by automatically generating valid test cases by mining API usages from real-world Android apps.

Why Dynamic Testing. We now present a concrete example to motivate why there is a need to generate more and better unit test cases for Android APIs to pinpoint compatibility issues. There is an Android API called *getNotificationPolicy()*, located in class *NotificationManager*. At the moment, there are no unit tests provided for this API. The lack of

[3]The APIs in `platform/frameworks/base` path.

API 23 (getNotificationPolicy Added at this version)

```
/**
 * Gets the current notification policy.
 *
 * <p>
 * Only available if policy access is granted to this package.
 * See {@link #isNotificationPolicyAccessGranted}.
 */
public Policy getNotificationPolicy() {
    INotificationManager service = getService();
    try {
        return
service.getNotificationPolicy(mContext.getOpPackageName());
    } catch (RemoteException e) {
    }
    return null;
}
```

API 24-27

```
/**
 * Gets the current notification policy.
 *
 * <p>
 * Only available if policy access is granted to this package.
 * See {@link #isNotificationPolicyAccessGranted}.
 */
public Policy getNotificationPolicy() {
    INotificationManager service = getService();
    try {
        return
service.getNotificationPolicy(mContext.getOpPackageName());
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

API 28 latest

```
/**
 * Gets the current user-specified default notification policy.
 *
 * <p>
 */
public Policy getNotificationPolicy() {
    INotificationManager service = getService();
    try {
        return
service.getNotificationPolicy(mContext.getOpPackageName());
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

Figure 5.1: The source code of Android API *getNotificationPolicy()*.

solid testing for this API has unfortunately led to various problems, as demonstrated by the various discussions on StackOverflow [Stack OverFlow, 2021], one of the most widely used question and answer websites.

Figure 5.1 presents the brief evolution of the source code of *getNotificationPolicy()*. This API was introduced to the Android framework at API level 23. The apps that accessed this API would crash on devices powered by Android with API level 22 or earlier, resulting in forwarding compatibility issues. During the evolution of the Android framework, the implementation of *getNotificationPolicy()* is quickly changed at API level 24 (i.e., no longer returns null). Nevertheless, at this point, the comments are not changed, suggesting potential compatibility issues because of changed semantics. At API level 28, this API is further changed. This time, only the comments are changed, i.e., the implementation of the API is kept the same. This change further suggests that this API may be involved with compatibility issues as the source code of the API and its comments are inconsistent at a certain period (over six API levels).

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3   super.onCreate(savedInstanceState);
4   setContentView(R.layout.activity_main);
5
6   NotificationManager mng = (NotificationManager) this.getSystemService("notification");
7   NotificationManager.Policy policy = mng.getNotificationPolicy();
8   int priorityCallSenders = policy.priorityCallSenders;
9 }
```

Listing 5.1: Code example of invoking API *getNotificationPolicy()*.

The actual implementation of *getNotificationPolicy()* is through the complicated inter-process communication mechanism (e.g., the API is defined via Android Interface Defini-

tion Language (AIDL). Thus, it is non-trivial to confirm if there is a compatibility issue after API level 23 by only (statically) looking at the Java code of the framework. It is still a known challenge to statically analyze Java and C code at the same time.

We resort to a dynamic approach to check if *getNotificationPolicy()* suffers from compatibility issues. Specifically, we implement a simple Android app with minimal lines of code to invoke the API (i.e., lines 6-7 in Listing 5.1) and also the usage of the return value of this API(i.e., lines 8 in Listing 5.1). The minimal and targeted SDKs of this app are set to be 21 and 30, respectively. We then launch this app on ten emulators covering Android API levels from 21 to 30. As expected, the app throws *NoSuchMethodError* as the API is not yet introduced at API level 21 and 22. From API level 23 to 27, the app throws *SecurityException* with the message "Notification policy access denied", due to a lack of declaration of Android permissions. Even though the permission was granted, the API can still introduce compatibility issues at API level 23 because the return value can be null, causing *NullPointerException* later on (e.g., lines 8 in Listing 1)). Surprisingly, since API level 28, the app does not throw any exception, even though no permissions are declared as well. We then go one step further to track the detailed implementation of *service.getNotificationPolicy(String)* and found that the enforcement of policy access (i.e., line 3 in Listing 5.2) is removed in the API at API level 28, which explains why the app no longer crashes since API level 28.

```
1  @Override
2  public Policy getNotificationPolicy(String pkg) {
3-   enforcePolicyAccess(pkg, "getNotificationPolicy");
4    final long identity = Binder.clearCallingIdentity();
5    try {
6     return mZenModeHelper.getNotificationPolicy();
7    } finally {
8     Binder.restoreCallingIdentity(identity);
9  }}
```

Listing 5.2: Code changes of method *getNotificationPolicy(String)*, which includes the underline implementation of Android API *getNotificationPolicy()*.

These observed runtime behaviours strongly indicate that the direct invocation of *getNotificationPolicy()* will very likely result in two compatibility issues: (1) a method is

not yet defined and (2) method semantics have been altered. Ideally, such issues – especially the latter case – should not be introduced into the Android framework. However, due to the lack of API unit tests, such issues are non-trivial to identify and avoid. It is hence essential to provide more and better unit test cases for all Android APIs. Since it is time-consuming to achieve this manually, we argue that there is a strong need to provide automated approaches to automatically generate unit test cases for Android APIs to identify compatibility issues as early as possible. In this work, we propose to generate unit test cases for Android APIs by learning from existing Android API usages.

Why mining API usage. In addition, generic test generation tools [Fraser and Arcuri, 2011a, Pacheco and Ernst, 2007, Kex, 2022] mainly targeting on satisfying coverage criteria for classes, while compatibility issues are mainly caused by the fast-evolving of APIs [Li et al., 2018b], which makes them insufficient in detecting compatibility issues. Specifically, such generic test case generation approaches (such as EvoSuite) are tailored to generate tests based on the source code of classes, lacking API usage knowledge [Kechagia et al., 2019], including both API calling context and API dependency knowledge. This information is crucial to setting up the environment for successfully calling Android APIs properly to detect compatibility issues. In addition, all of these tools completely ignore semantic-level behaviours at the API level, leading to many compatibility issues undetected. Especially in Android, APIs often come with usage caveats, such as constraints on call order [Ren et al., 2020]. Thus, it is essential to capture API dependencies [Zhang et al., 2014] involved in the calling context before invoking the target API. However, it is challenging for traditional test generation tools to achieve this since they generate test suites only based on source code, which lacks API dependency knowledge. Thus, the insufficiency of the generic coverage-based test case generation approach motivates us to mine API usage to generate much more effective test cases in detecting compatibility issues.

5.3 Approach

The main goal of this work is to automatically generate unit tests for the Android framework to provide better coverage at the unit test stage of as many Android APIs as possible. Fig. 5.2 outlines the process of JUnitTestGen, which is made up of two modules involving a total of nine steps. We first locate target API invocations after disassembling the APK bytecode. We then apply inter-procedural data-flow analysis to identify the API usage, including API caller instance inference and API parameter value inference. We then execute these generated test cases on Android devices with different Android versions (i.e., API levels). The following elaborates on the detailed process of each module.

The first module, *Automated API Unit Test Generation*, includes five sub-processes to automatically generate unit test cases for Android APIs. This module takes as input an Android API (or an API list) for which we want to generate unit tests and an Android app that invokes the API and outputs a minimum executable code snippet involving the given API. This code snippet is the API's unit test case.

The second module, *App Compatibility Testing*, includes two sub-processes. This module takes as input the previously generated unit test cases to build an Android app, allowing direct executions of the test cases on Android devices running different framework versions (i.e., API levels). The output of this second module is the execution results of the test cases concerning different execution environments. Using this, JUnitTestGen can then determine all the Android APIs suffering from potential compatibility issues. We now detail these two modules of JUnitTestGen below.

5.3.1 Automated API Unit Test Generation

As shown in Figure 5.2, the first module of JUnitTestGen, namely Automated API Unit Test Generation, is made up of five steps.

I.1 Android Bytecode Disassembling. JUnitTestGen takes Android APKs as input.

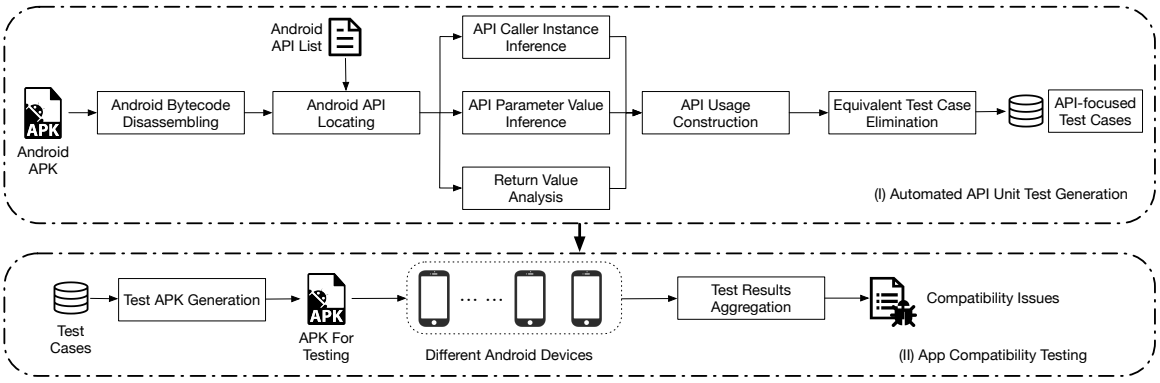


Figure 5.2: The working process of our approach.

Given an Android app, the first step of JUnitTestGen is to disassemble the Dalvik bytecode to an intermediate representation. In this work, we leverage Soot [Vallée-Rai et al., 2010], which provides precise features for code analysis. In particular, it supports 3-address code intermediate representation Jimple and accurate call-graph analysis framework Spark[Lhoták and Hendren, 2003]. On top of Soot, JUnitTestGen is able to convert Android APK's bytecode into Jimple precisely.

I.2 Android API Locating. The second step towards locating the practical usages of Android APIs is quite straightforward. JUnitTestGen visits every method of each application in its Jimple representation, i.e., statement by statement, to check if any of the APIs in the input list is invoked. If so, we record the location and mark the usage of the API as located.

I.3 API Caller Instance Inference. Android APIs are invoked as either *static methods* (also known as *class methods*) or *instance methods*. A static method can be called without needing an object of the class, while an instance method requires an object of its class to be created before it can be called. Listing 5.3 shows an example for each method type in the Jimple representation. The static method (*boolean equals()*) can be called directly as long as the API signature (<*android.text.TextUtils: boolean equals(java.lang.CharSequence, java.lang.CharSequence)*>) is acquired, which has already been done in step I.2. How-

ever, to invoke the instance method (*void setLayoutParams()*), its calling object *$r5* needs to be identified. Specifically, a backward data-flow analysis is needed to locate both the definition statement of *$r5* and any intermediate APIs (or methods) on the call trace that may change the behaviour of the caller instance.

```
1 //Calling a Static API
2 <android.text.TextUtils: boolean equals(java.lang.CharSequence,java.lang.CharSequence)>
3
4 //Calling an Instance API
5 $r5.<android.widget.ListView: void
      setLayoutParams(android.view.ViewGroup$LayoutParams)>
```

Listing 5.3: Examples of calling static and instance APIs.

Identifying the calling object and constructing the call trace is a non-trivial task. As the calling object's instantiation could depend on multiple method calls (e.g., can be defined in other methods and passed on via callee's returned values), the caller instance inference process needs to be inter-procedural. Furthermore, each of the involved methods in the instantiation process may further require specific parameter values, which need to be properly prepared in order to successfully create the calling object. Hence, the caller instance inference process requires backtracing not only the direct caller instance but also many other variables leveraged by the app to instantiate the calling object.

Algorithm 1 API Caller Instance Inference.

Input: api_t: the target API
Output: M: a list of method invocations towards api_t
1: **function** CALCULATEMINIMUMEXECUTABLECONTEXT(api_t)
2: $M = \emptyset$
3: $stmt_t \leftarrow$ statement contains api_t
4: **if** api_t is a static API **then**
5: $M \leftarrow$ The API signature of api_t
6: **end if**
7: **if** api_t is an instance API **then**
8: $assignStmt \leftarrow$ GETDEFINITIONSTMT($stmt_t$)
9: $M \leftarrow$ CONSTRUCTCALLTRACE($assignStmt, \emptyset$)
10: **end if**
11: **return** M
12: **end function**

Algorithm 1 gives the details of the approach. Given a target API as input, we apply a backward data flow analysis to identify the minimum executable context of the target API (lines 1~12). As shown in Algorithm 1, we describe the API caller inference process for both static methods (lines 4~6) and instance methods (lines 7~10). For static meth-

Algorithm 2 Construct Call Trace.

Input: $stmt_{co}$: A definition statement of the calling object
$visitedStmts$: The set of statements visited by the analysis.
Output: $callTrace$: a list of method invocations towards $stmt_{co}$

```
1: function CONSTRUCTCALLTRACE(stmt_co, visitedStmts)
2:     callTrace ← ∅
3:     if (stmt_co in visitedStmts) OR (stmt_co is Constant) OR (stmt_co is Syetem API) then
4:         return callTrace
5:     end if
6:     visitedStmts ← visitedStmts ∪ stmt_co
7:     if stmt_co is ParameterRef then
8:         stmt_inv ← getInvolvingStmtsfromCallGraph(stmt_co)
9:         for each s ∈ stmt_inv do
10:            definitionStmt_param ← GETDEFINITIONSTMT(s)
11:            callTrace ← CONSTRUCTCALLTRACE(definitionStmt_param, visitedStmts)
12:        end for
13:    end if
14:    if stmt_co is InvokeExpr then
15:        stmt_inv ← getInvolvingStmtsfromCallGraph(stmt_co)
16:        for each s ∈ stmt_inv do
17:            method_host ← the host method of s
18:            returnStmt ← getReturnStmtFromMethod(method_host)
19:            definitionStmt_returnStmt ← GETDEFINITIONSTMT(returnStmt)
20:            callTrace ← CONSTRUCTCALLTRACE(definitionStmt_returnStmt, visitedStmts)
21:        end for
22:    end if
23: end function
```

ods, we return its corresponding API signature, which has been obtained in step I.2. For instance methods, we first locate the definition statement of the calling object through invoking method *getDefinitionStmt* (line 8), which returns the definition statement for a local variable. This method walks the inter-procedural control flow graph from the target API statement in reverse order, aiming to look for the nearest assignment statement defining the API's calling object. After that, with the help of the function *constructCallTrace*(i.e., defined in Algorithm 2) (line 9), we can extract the call trace corresponding to the calling object. As shown in Algorithm 2, we handle parameter callers (lines 7~13) and method callers (lines 14~22), respectively. If the definition statement of the calling object comes from a parameter reference, we first retrieve all the statements at which the invocation occurs, and then for each of the statements, we recursively construct its call trace by calling the method *constructCallTrace*. A similar process has been applied to handle method callers (lines 14~22), which recursively construct the call trace involving statements of the calling object. The recursive process will not terminate until any of the conditions have been satisfied in line 3, i.e., either the statement is a constant, or it has been visited before,

or it is an Android system API.

We elaborate on this process with a Jimple code example presented in Listing 5.4. In this example, the target Android API to test (i.e., *queryDetailsForUid(int,String,long,long,int)*) is invoked in line 26 by the calling object *$r3*, where *$r3* is a returned value of a self-defined method *getNetworkStatsManager(Context)* (line 22). We then step into the definition of the method *getNetworkStatsManager(Context)* (lines 1~7), and further backtrace the variables *$r2* and *$r1* along the call chain. *$r1* retrieves the network stats service from the application context *$r0* (line 4), and finally, the backtrace terminates at *$r0* (line 3), where all unknown variables are resolved.

```
1  public static android.app.usage.NetworkStatsManager
       getNetworkStatsManager(android.content.Context)
2  {
3   $r0 := @parameter0: android.content.Context;
4   $r1 = $r0. getSystemService("netstats");
5   $r2 = (android.app.usage.NetworkStatsManager) $r1;
6   return $r2;
7  }
8
9  public static int getUid(android.content.Context) throws
       android.content.pm.PackageManager$NameNotFoundException
10 {
11  $r0 := @parameter0: android.content.Context;
12  $r1 = $r0.getPackageManager();
13  $r2 = $r0.getPackageName();
14  $r3 = $r1.getApplicationInfo($r2, 1);
15  i0 = $r3.<android.content.pm.ApplicationInfo: int uid>;
16  return i0;
17 }
18
19 public static float getCurAppFlow(android.content.Context) throws
       android.content.pm.PackageManager$NameNotFoundException
20 {
21  $r0 := @parameter0: android.content.Context;
22  $r3 = DeviceInfoUtil.getNetworkStatsManager($r0);
23  $l1 = System.currentTimeMillis();
24  $i0 = DeviceInfoUtil.getUid($r0);
25  //Target API
26  $r5 = virtualinvoke $r3.<android.app.usage.NetworkStatsManager:
       android.app.usage.NetworkStats
       queryDetailsForUid(int,java.lang.String,long,long,int)>(0, "", 0L, $l1, $i0);
27 }
```

Listing 5.4: Code example demonstrating the usage of API *queryDetailsForUid*. The code snippet is extracted from app *com.eyoung.myutils*.

In addition to inter-procedural data-flow analysis, JUnitTestGen also needs to be field-aware. When performing backward data-flow analysis, the access of fields may break the original flow and hence may lead to unexpected results if not properly handled.

To mitigate this, we transform a field involved in the call trace of the API under testing (in the analyzed app) into a local variable in the generated test case. The local variable will be initiated following the same method as it is assigned in the original app. We then search the whole class to check how the field's value is assigned and subsequently apply the same method to initialize the local variable. If we cannot find the field's assignment or the assigned value is complicated to be reconstructed, we will use a dummy object to mock the required value.

In addition, JUnitTestGen needs to handle branches in the backward dataflow analysis. Specifically, we leverage the Inter-procedural Control Flow Graph (ICFG [Bodden, 2012]), which is a combination of call-graph(CG) and control flow graph(CFG), to identify the minimum executable context of the target API. Here, CG is a graph representing the calls between methods over the entire program, while CFG is a graph that represents the control flows in a single method. ICFG treats each statement (or a set of sequential statements) as a node, including branch statements that enable path-sensitive analyses, i.e., the propagation of different information along different branches. With ICFG, we are able to implement branch analysis by analyzing the structure of the graph. To this end, for those methods involving multiple branches, JUnitTestGen will separate each branch to form a different test case.

I.4 API Parameter Value Inference. To support API compatibility testing, e.g., to ensure that the API will, in any case, be reached once the test case is executed, we propose to directly assign possible values to the API's parameters inside the test case, i.e., the test case per se will not contain any parameter. To achieve this, JUnitTestGen also infers possible values for each parameter of the API to be tested. This step follows the same strategy i.e., the approach adopted to infer API caller instances, to infer the API's parameter values. This is done by performing inter-procedural backward data-flow analysis. We apply the same algorithms described in Algorithm 1 and Algorithm 2 on each parameter object

to figure out the exact value.

Unfortunately, some Android APIs' parameter values may involve sophisticated operations when building their run-time values that are non-trivial to correctly retrieve statically. Specifically, if the analysis process does not end up at a constant/Android system API statement, the value of a parameter is regarded as being undiscovered. To mitigate this, we introduce a set of pre-defined rules to generate dummy values for such APIs that have their parameter values hard to retrieve practically. Some of the representative rules for generating dummy values for such hard-to-retrieve parameters are:

- For the eight primitive data types in Java (such as int, double, etc.) or their wrapper data types (such as Integer, Double, etc.) – we provide random values for each of them that conform to their types.
- For the String data type in Java – we generate a random alphanumeric string.
- For the Array parameter whose basic type is the eight primitive data types (or their wrapper data types) in Java (such as int, Integer, etc.) – we generate an Array variable with random primitive values.
- For Android system-related objects (or the intermediate objects in the calling object's instantiation process), we use a heuristic approach to obtain the corresponding constructors to create their instances. If an object has multiple constructors, we select the simplest one (with the least number of parameters) to achieve the highest possibility of constructing a valid object.

I.5 Return Value Analysis. To support detecting compatibility issues caused by return values (e.g., a given API may return A at API level X and B at API level Y), we propose to output the return value of the target API at the end of the test case. To achieve this, JUnitTestGen adds a statement at the end of the test case to further record the API's return value.

I.6 API Usage Construction. After obtaining the caller instance, the invocation statements along the call trace and the return object, we can now recover the call sequence from program entry to the target Android API by reversing the retrieved statements step by step. Based on the results of API caller instance inference along, JUnitTestGen will generate a test case containing the same number and type of parameters as the API to be tested. For example, as shown in Listing 5.5 at line 3, the generated test case contains five parameters, in the same type and order of the API under testing.

```
1 //for supporting generic testing
2 @Test
3 public void testQueryDetailsForUid(int var1, String var2,long var3, long var4, int
      var5) throws Exception {
4  Context var6 = InstrumentationRegistry.getTargetContext();
5  Object var7 = var6.getSystemService("netstats");
6  NetworkStatsManager var8 = (NetworkStatsManager) var7;
7  var8.queryDetailsForUid(var1, var2, var3, var4, var5);
8 }
9
10 //for supporting compatibility testing
11 @Test
12 public void testQueryDetailsForUid() throws Exception {
13  long var1 = System.currentTimeMillis();
14
15  Context var2 = InstrumentationRegistry.getTargetContext();
16  PackageManager var3 = var2.getPackageManager();
17  String var4 = var2.getPackageName();
18  ApplicationInfo var5 = var3.getApplicationInfo(var4, 1);
19  int var6 = var5.uid;
20
21  NetworkStats var7 = testQueryDetailsForUid(0, "", 0L, var1, var6);
22 // Output return value
23  out(var7);
24 }
```

Listing 5.5: Examples of the generated test cases for API *queryDetailsForUid (int networkType, String subscriberId, long startTime, long endTime, int uid)*.

Taking the results of the API parameter value inference step, JUnitTestGen will generate another test case containing no parameters (line 10 in Listing 5.5). This test case will directly call the former test case with prepared parameter values. This test case is specifically designed to support API compatibility testing. The former test case, on the contrary, is designed to serve a more general purpose. With the help of fuzzing testing approaches (to generate possible parameter values for the test case), we expect the former test case could be leveraged to discover not only compatibility issues but also design defects such as bugs and security issues. This trade-off allows JUnitTestGen to generate test cases that

are at least suitable for identifying signature-based compatibility issues (e.g., a given API is no longer available in a certain framework), although it may not be effective enough to help identify semantic change involved compatibility issues.

Please note that there are several special classes, such as *InstrumentationRegistry*, involved in the generated unit test cases. These classes are part of the Android Testing Support Library provided by Google for supporting instrumented unit tests. Compared to traditional unit tests, also known as local unit tests which can run on the JVM, instrumented unit tests require the Android system to run (e.g., through physical Android devices or emulators). Since this requires us to generate actual Android apps to run on Android devices or emulators, instrumented tests are much slower than local unit tests.

Nevertheless, we still choose to use instrumented tests to examine the compatibility of Android APIs. This is because instrumented tests provide more fidelity than local unit tests, which we have found is essential to reveal potential compatibility issues, especially those that involve device-specific issues.

I.7 Equivalent Test Case Elimination. Considering that the constructed test cases could be equivalent (i.e., duplicated), it is necessary to filter them out to save subsequent testing time and resources. Here, based on the concept of semantic equivalence defined in operational semantics [Jiang and Su, 2009], we consider that two test cases are equivalent if they share the same API invocation sequence. To this end, we first obtain the API invocation sequence for each test case and then examine the discrepancy between any two of them to check if $O_a = O_b$, where O_a and O_b are the lists of API invocations in two different test cases. Based on this rule, we are able to eliminate equivalent tests (i.e., the first test case is retained). After this step, for the sake of simplicity, if there are still multiple test cases retained for a given API, we will select the small-scale one (with the least number of method invocations) for supporting follow-up analyses.

5.3.2 App Compatibility Testing

Using the unit test cases generated by the first module, the second module of JUnitTestGen leverages them to check if the corresponding Android APIs will likely induce app compatibility issues. It first assembles all the test cases into an Android app and then aggregates their execution results against different devices running different Android frameworks. We now briefly detail these two steps below.

II.1 Test APK Generation. As discussed earlier, we have to resort to instrumented unit tests to examine Android APIs' incompatibilities. This process essentially requires us to generate an Android app (or APK) to be installed and executed on Android systems. Fortunately, Google has provided such a mechanism to achieve this purpose, i.e., supporting instrumented tests for a limited number of Android APIs. In this work, we directly reuse this mechanism to generate the test APK for all the unit tests automatically generated by JUnitTestGen.

II.2 Test Results Aggregation. After the test APK is generated, we can distribute it for execution on multiple devices. Since the test APK contains only known test cases, it is quite straightforward to execute it fully. The only challenge that lies in this step is to select the right set of devices on which to execute the test cases to reveal as many incompatible APIs as possible. Crowdsourced app testing could be an approach to achieve this purpose.

After installing and executing the generated test APK on multiple devices, the last step is in aggregating the test results to highlight potential compatibility issues in the app. Inspired by the experimental setup of the work proposed by Cai et al. [Cai et al., 2019], we consider an API as a potential incompatible case if 1) its corresponding test case can successfully run on a nonempty set of devices while failing on others; 2) its corresponding test case returns different values when running on different SDKs.

5.4 Evaluation

Our JUnitTestGen aims to generate unit test cases covering as many Android APIs as possible, so as to allow the discovery of more API-induced compatibility issues in apps. To evaluate if this goal has been fulfilled, we propose to answer the following three research questions.

RQ1 To what extent can JUnitTestGen generate executable unit test cases for Android APIs?

RQ2 How effective is JUnitTestGen in discovering API-induced Compatibility Issues?

RQ3 How does JUnitTestGen compare with existing tools in detecting compatibility issues?

5.4.1 Experimental Setup

To investigate the success rate of JUnitTestGen in producing valid test cases, we randomly select 1,000 Android apps for each target SDK version between 21 (i.e., Android 5.0) and 30 (i.e., Android 11.0[4]) from AndroZoo to prepare the experimental dataset. Here, we select 1,000 apps for each target SDK version because compatibility issues mainly lie in the evolution of APIs on different Android SDK versions [Li et al., 2018b]. Here, the criteria for app selection are based on the targetSdkVersion, which is the most appropriate API level on which the app is designed to run. Hence, the overall dataset for the experiment contains 10,000 Android apps whose target API versions are distributed equally across ten API levels. Our experiment runs on a Linux server with Intel(R) Core(TM) i9-9920X CPU @ 3.50GHz and 128GB RAM. The timeout setting for analyzing each app with JUnitTestGen is 20 minutes. In this experiment, we generate the test cases for all Android APIs that have been invoked in the apps. However, users of JUnitTestGen could provide a customized list

[4]The latest version at the time when we conducted this study.

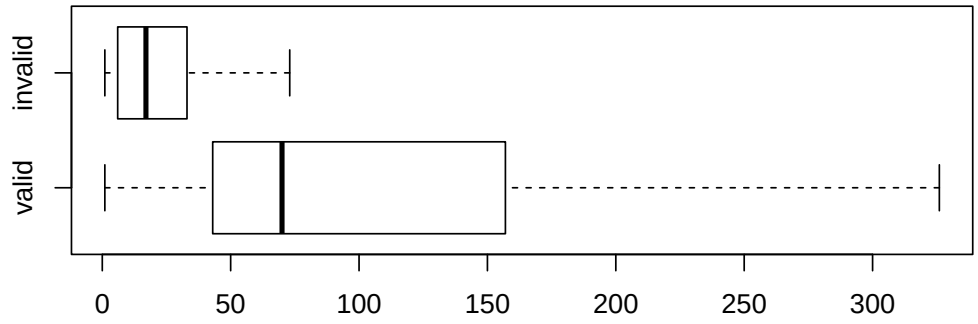


Figure 5.3: Distribution of the number of valid and invalid test cases per APK.

of APIs to only generate test cases based on their interests.

5.4.2 RQ1-Effectiveness

Our first research question concerns the effectiveness of JUnitTestGen in mining API usages from existing Android apps to generate valid unit tests for Android APIs. In this work, we consider a test case to be valid if (1) the generated code snippet can be successfully compiled on all API levels and (2) the test case does not throw an exception before the execution point of the API on all API levels. The first condition ensures that the test case is syntactically correct. The second condition makes sure that the API's execution environment is properly set up. In other words, the second condition ensures that the exceptions we collected from valid test cases are exceptions thrown by the API under testing, which is essential for examining if the API will induce compatibility issues.

Result. Among the 10,000 randomly selected apps, JUnitTestGen generates in total 1,032,182 test cases. After eliminating the equivalent test cases, 66,499 of them are retained as distinct test cases, w.r.t. 28,367 distinct Android APIs. For the sake of simplicity, we select the small-scale test case (with the least number of invocation sequences) for each API (i.e., 28,367 test cases) for further study. By compiling and executing these 28,367 test cases, we further confirm that 5,562 of them are invalid (i.e., 22,805 of them are valid), giving a success rate of 80.4% in generating valid test cases. In addition, our manual

analysis on 100[5] randomly selected test cases confirm that these test cases generated by JUnitTestGen are indeed valid.

Figure 5.3 summarizes the distribution of the number of valid and invalid test cases generated per app. The median number of valid and invalid test cases generated per app are 70 and 17, respectively, while their average are 106.29 and 25.37, respectively. Like most other state-of-the-art approaches, even though our static analysis approach has limitations so that it cannot generate valid test cases for every API, especially the complex ones, **our approach is still capable of generating more valid test cases than invalid ones.** This demonstrates the effectiveness of our approach in mining Android API usages to generate API unit test cases.

We note that the success rate of generating valid test cases – 80.4% at the moment – is important but not crucial to our work. Theoretically, as long as we increase the number of Android apps considered for learning, we would likely be able to generate valid test cases for the given API under testing. For the test cases that are regarded as invalid, we further manually look into their root causes. Our in-depth analysis reveals that the invalid cases are mainly caused by the lack of prerequisites (e.g., resource files), especially in UI-related APIs. For example, UI objects can hardly be programmatically initialized without certain resource files.

We further look at the efficiency of JUnitTestGen by reporting its time cost when applied to generate test cases. Figure 5.4 summarizes the distribution of time consumed by JUnitTestGen per app. On average, it takes 17.45 seconds to process an app. The median time cost is 14 seconds. The fact that the time spent by JUnitTestGen to process an Android app is quite small suggests that it is practical to apply JUnitTestGen to analyze large-scale Android apps.

[5]The sample size is determined based on a confidence level at 95% and a confidence interval at 10(`https://www.surveysystem.com/sscalc.htm`).

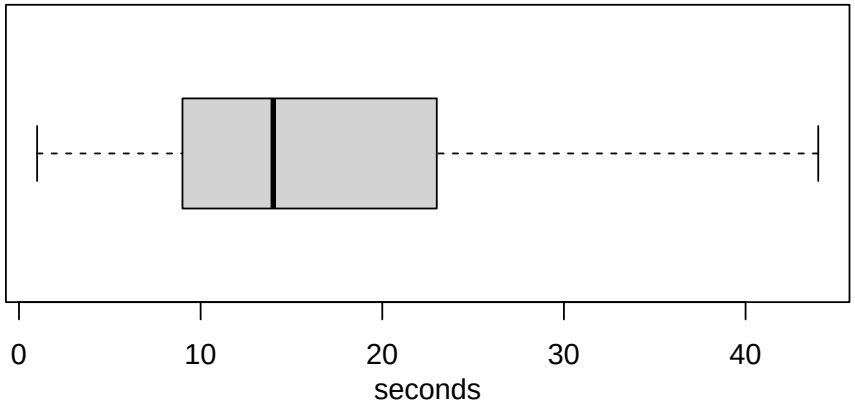


Figure 5.4: Distribution of time spent by JUnitTestGen to generate test cases per APK.

Answers to RQ 1 *By learning from existing API usages, our approach can automatically generate API unit test cases. Despite various challenges posed by advanced Android programming features, our static analysis approach can still achieve over 80% of the success rate in generating valid test cases.*

5.4.3 RQ2-Performance on Real-World Applications

The ultimate goal of JUnitTestGen is to help better identify API compatibility issues that occur during Android system evolution. To this end, in this research question, we evaluate, based on the generated valid unit test cases, to what extent can JUnitTestGen help in identifying API-induced compatibility issues.

We consider an Android API has a compatibility issue if the execution results on different Android SDK versions are inconsistent. More specifically, an API is considered to have a compatibility issue if any of the following happens: (1) the corresponding test case runs successfully on some Android SDK versions but fails (e.g., throws errors or exceptions) on others; or (2) the corresponding test case throws different errors or exceptions on different Android SDK versions (e.g., throws *NoSuchMethodError* on some versions, while throws *IllegalArgumentException* on others); or (3) the return values of a target API are non-identical on different SDK versions.

Result. Recall that JUnitTestGen successfully generates 22,805 distinct valid test cases

covering 22,805 unique Android APIs, based on the randomly selected 10,000 apps. In this work, these 22,805 test cases are respectively executed on ten Android emulators running API levels from 21 to 30.

By comparing the experimental results against the aforementioned three rules, **we are able to locate 3,488 compatibility issues covering 2,695 Android APIs.** Note that some APIs may involve more than one compatibility issue. To confirm whether the APIs identified by JUnitTestGen indeed have compatibility issues, we manually examined 100 randomly selected APIs reported to have compatibility issues, 100 of them are confirmed to have compatibility issues (i.e., true positive results). Here, we remind the readers that all of the compatibility issues are actually identified through dynamic analysis, which is expected to be highly accurate. In the manual validation process, we manually examine the APIs' implementation in Android framework source code across different SDK versions and compare it with the release notes in the official API documentation.

According to the root causes of the compatibility issues, we further categorize them into the following types.

- **Type 1: Signature-based compatibility issue.** This type refers to the incompatibility caused by adding new APIs, deprecating existing APIs, or changing existing APIs' signatures, such as changing parameters or return types.

- **Type 2: Semantics-based compatibility issue.** This type refers to the incompatibility caused by the same API (i.e., the signature is not changed) behaving differently on different Android API levels. Based on the APIs' behaviours, semantic-based compatibility issues are further breakdown into the following two sub-types.

  1) Type 2.1 Semantics-based compatibility issue: Different Errors: The APIs categorized into this type will throw exceptions or errors (including app crashes) on devices running some SDK versions but will not (either running successfully or throw

different exceptions/errors) when running on devices with other SDK versions.

2) Type 2.2 Semantics-based compatibility issue: Different Return Values: APIs of this type will not directly cause compatibility issues. Under the same experimental setting, these APIs will return different values when running on different SDKs. However, if the different return values are not specifically distinguished, the subsequent code that uses these return values may behave differently on different devices, leading to also compatibility issues.

Among the 3,488 identified compatibility issues, 438 of them suffer from signature-based issues, while 3,050 are caused by semantic issues. Within the semantic-based compatibility issues, 946 of them are caused by different errors (Type 2.1), while the remaining 2,104 cases observe different return values (Type 2.2). Table 5.1 summarizes the possible errors/exceptions that cause the signature/semantic compatibility issues.

As expected, the most common error is no such method error, which can be caused by (1) the API being deprecated and removed from the framework, or (2) the API not yet introduced. Both reasons are introduced by the evolution of the framework. As also revealed by Li et al. [Li et al., 2018b], the fast evolution of the Android framework has indeed introduced a lot of APIs that will likely induce compatibility issues. Except for signature-based issues, which are relatively easy to be statically identified (for example, by comparing the framework codebase of different versions), our approach has also found 2,974 issues (over five times the number of Type 1 issues) involving semantic changes of APIs, which are non-trivial to be identified statically [Liu et al., 2021a].

Below, we elaborate on real-world compatibility issues for each type of case study.

Case Study 1: Signature-based Compatibility Issue. The API *android.content.pm.LauncherApps#h*
has been reported to contain a signature compatibility issue. The corresponding test case*(whose API usage is extracted from app ch.deletescape.*

Table 5.1: Categories and statistics of the observed error/exception types associated with compatibility issues.

Issue Type	Errors/Exceptions	Count
Signature	NoSuchMethodError	270
	NoClassDefFoundError	163
	NoSuchFieldError	5
Semantic	SecurityException	196
	NullPointerException	189
	RuntimeException	139
	Resources$NotFoundException	113
	IllegalArgumentException	67
	NoSuchElementException	42
	Crash	41
	IllegalStateException	24
	AndroidRuntimeException	23
	IOException	15
	ArrayIndexOutOfBoundsException	15
	FileNotFoundException	12
	PackageManager$NameNotFoundException	10
	IndexOutOfBoundsException	10
	ClassCastException	9
	IllegalAccessError	9
	ActivityNotFoundException	8
	StringIndexOutOfBoundsException	5
	UnsupportedOperationException	5
	BadTokenException	3
	CanceledException	3
	ErrnoException	2
	KeyChainException	2
	SQLiteCantOpenDatabaseException	1
	ParseException	1
	StackOverflowError	1
	NumberFormatException	1

lawnchair.plah[6]*)* throws *NoSuchMethodError* on Android SDK version 21 to 23 but can be successfully executed on Android SDK version 24 to 30. This result suggests that the API was introduced to the Android system since API level 24; therefore, it would cause an error if the containing app runs on devices with Android SDK version earlier than 24. However, according to the official Android API documentation, this API was added in API

[6]SHA-256:bf4e6e7fb594cd9db4b168a68f70157ad9c1fea0192e0bd5d9a39d1c38802639

level 25 [Google, 2021a], which is imprecise according to our result. We further checked the source code of Android SDK 24 and confirmed its existence.

Case Study 2: Semantics-based compatibility issue caused by different Exceptions. The API *android.app.NotificationManager #notify* has been reported to contain a semantic compatibility issue. The corresponding test case*(whose API usage is extracted from app com.ag.dropit*[7]*)* can be successfully executed on Android SDK version 21 to 22 but throws *IllegalArgumentException* on Android SDK version 23 to 30. We manually looked into its source code in the Android codebase and found that the actual implementation of this API has been changed since API level 23, which added a sanity check of object *mSmallIcon*. This explains why it throws *IllegalArgumentException* when there is no valid small icon from the API level after 23. Unfortunately, the official Android API documentation does not reflect this change, which is misleading.

Case Study 3: Semantics-based compatibility issue caused by different return values. The API *(extracted from app cleaner. clean.booster*[8]*)* <*android.text.format.Formatter: String formatShortFileSize(Context, long)*> has a return value-induced compatibility issue. The format of the return values vary on different API levels: given the 1L(The long data type of value 1) as the second parameter, the return value on API level 21 to 22 is *"1.0B"*, the return value on API level 23 is *"1.0 B"* (with additional whitespace between 1.0 and B), while the return value on API level 24 to 30 is *"1 B"*. The difference in return values can introduce potential issues if app developers rely on the return value to implement future functions without checking the running API level. For example, if app developers cast the return value from String to Byte afterwards, it may throw an exception if users ignore the value discrepancy on different API levels.

[7] SHA-256:30f7f72cebeffd7c6e26489198ee5ad244bd44b076dd9cb59865d8b0e82a86af
[8] SHA-256:def5db37b3a68de62a0472e872700092060bdec3e875d4f476fcda52795bceb2

Table 5.2: The comparison results between JUnitTestGen and Evosuite, CiD.

Tool	# Type 1	# Type 2.1	# Type 2.2
JUnitTestGen	438	946	2,104
Evosuite	36	0	0
CiD	864	-	-

Answers to RQ 2 *Our approach is useful in automatically pinpointing API-induced compatibility issues. It also goes beyond the state-of-the-art to be capable of detecting not only signature-based compatibility issues but also more significant semantics-based compatibility issues, i.e., the corresponding APIs' signatures are kept the same, while their semantics are altered.*

5.4.4 RQ3-Comparison with State-of-the-art

Considering the main purpose of our work is generating test cases for detecting compatibility issues, both generic test case generation approaches, such as EvoSuite [Fraser and Arcuri, 2011b] and compatibility issues detection tools, such as CiD [Li et al., 2018b], are selected as the baselines to evaluate our approach. We evaluate the performance of JUnitTestGen, EvoSuite [Fraser and Arcuri, 2011b] and CiD [Li et al., 2018b] in detecting compatibility issues. Overall, table 5.2 lists the number of compatibility issues found by JUnitTestGen, EvoSuite and CiD. We then break down the comparative results as follows:

Comparison with EvoSuite. To compare JUnitTestGen with generic test case generation tools, we choose EvoSuite as the baseline because EvoSuite has been considered the state-of-the-practice test generation tool, which achieved the highest score at the SBST 2021 Tool Competition [Vogl et al., 2021]. EvoSuite uses an evolutionary search approach to generate and optimize test suites toward satisfying an entire coverage criterion for Java classes. We remind the readers that the objective of EvoSuite is to generate tests for classes, not directly aiming at generating tests for APIs, which are the main target when concerning compatibility issues happening in Android apps. Since Evosuite can only generate tests based on source code, we resort to AOSP from SDK 21 to 30 for Evosuite to generate test cases. In total, Evosuite successfully generates 5,335 test cases. We then execute all of them on SDK versions from 21 to 30 and apply the same rules for determining compatibil-

ity issues as used in JUnitTestGen.

As shown in Table 5.2, EvoSuite only finds 36 signature-based compatibility issues, and no semantic compatibility issues are identified. We further manually check the test cases generated by EvoSuite and observe that the false negatives (compared with JUnitTestGen) are mainly caused by overlooking API dependency information. For example, some APIs can only be invoked by system services, which makes EvoSuite fail to generate valid tests without knowing the usage of such APIs. Missing API dependency information makes EvoSuite insufficient in pinpointing compatibility issues. In other words, our comparison result reveals that mining API usage from apps is beneficial for finding compatibility issues.

Comparison with CiD. To the best of our knowledge, no work has been devoted to detecting compatibility issues in Android apps dynamically. CiD [Li et al., 2018b] is the closest work to ours on detecting compatibility issues. CiD model and compare API signatures on different SDK versions to detect compatibility issues. We thus evaluate the performance of JUnitTestGen and CiD on the same dataset in RQ1, which contains 10,000 Android apps.

In total, CiD detects 864 compatibility issues, as highlighted in Table 5.2. Out of the 864 compatibility issues detected by CiD, JUnitTestGen successfully identified 3,050 cases that have been overlooked by CiD. We further randomly analyze 50 false negatives of CiD and find that these issues are caused by the lack of semantics analysis of the API implementation. Specifically, when analyzing the evolution of APIs, CiD only examines the change of API signatures (including name, type, and parameters); hence it is not capable of detecting APIs that modify the implementation details but retain the same signature. Also, we find other 51 false positives of CiD are caused by imprecisely extracting the usage of the APIs, due to the context-insensitive approach of CiD when building the conditional call graph (CCG). On the other hand, we find that JUnitTestGen miss 375 cases reported in CiD (i.e., false negatives in JUnitTestGen). This is mainly caused by the sophisticated usage of

some APIs, which cannot easily be initialized programmatically (e.g. UI-related APIs). For example, some APIs may involve the initialization of UI objects that cannot be initialized programmatically. This makes it very challenging to automatically generate unit test cases in some circumstances. However, we argue that this limitation can be alleviated by manually adding prerequisite resources to create more valid tests. Overall, the comparison results reveal the weakness of static analysis approaches in detecting semantic compatibility issues, i.e., yields false-positive results and is hard to detect issues involving semantic changes in methods. It also demonstrates that our approach can indeed find more diverse compatibility issues and hence is promising to complement existing static approaches.

Answers to RQ 3 *JUnitTestGen outperforms the state-of-the-practice test generation tool, EvoSuite, and the state-of-the-art static compatibility detection tool, CiD, in pinpointing compatibility issues caused by the fast evolution of Android APIs. This experimental evidence shows the necessity to perform dynamic testing to pinpoint compatibility issues in Android apps, and it should take API usage dependencies into consideration when generating test cases to fulfill the dynamic testing approach.*

5.5 Discussion

We now discuss the potential implications and limitations of this work.

5.5.1 Implications

Better Supplementing Compatibility Analysis: JUnitTestGen is able to automatically generate tests for Android APIs for detecting both signature-based and semantics-based compatibility issues. Previous static analysis works [Li et al., 2018b] overlooked the semantics-based ones (i.e., APIs have the same signature but different implementation), which are more challenging to detect statically. Together with the state-of-the-art static analysis-based methods, our proposed method can provide a more comprehensive overview of compatibility issues in Android APIs. Therefore, we argue that there is a need to invent hybrid approaches to take advantage of both static analysis and dynamic analysis to conquer compatibility issues.

Beyond Compatibility Testing: JUnitTestGen is not only useful in pinpointing compatibility issues in Android APIs but also could be easily adopted to automatically generate test cases for other purposes. For instance, in the cases that an API takes various parameters as input, it can work with other testing approaches such as fuzz testing to explore the API's implementation dynamically. It hence goes beyond compatibility testing and provides a more general-purpose form of API testing.

Go Beyond Android. Our approach performs static program analysis to learn and generate test cases from existing API usages, which are not strongly attached to Android apps. We believe it could be easily adapted to analyze other Java projects, e.g., to automatically generate test cases for popular Java libraries. Although our approach cannot be directly applied to analyze projects written in other programming languages than Java, we believe the idea and methodology proposed in this paper could still work. We plan to explore these research directions in our future work. We also encourage our fellow researchers to explore this direction further.

5.6 Limitations

The main limitation of JUnitTestGen lies in its backward data-flow analysis when inferring API caller instance and API parameter values. Indeed, as already known in the community, it is non-trivial to implement a sound data-flow analysis. Other researchers often accept trade-offs to obtain relatively good results, and this is the same in our case. When the variables to be backwardly retrieved are complex, e.g., involving constructing various intermediate objects, JUnitTestGen rely on their simplest constructor to initialize the objects to generate a valid test case. Unfortunately, some of the constructors are too complex to initialize, leading to invalid test cases. Some other APIs may involve the initialization of UI objects that cannot be initialized programmatically (hence random values are leveraged to handle such cases). This makes it very challenging to automatically generate unit test

cases in all circumstances.

Second, currently, our JUnitTestGen data-flow analysis is agnostic to some advanced programming features, such as reflective and native calls. This may further impact the soundness of our approach. As part of our future work, we plan to integrate approaches developed by our fellow researchers to mitigate those long-standing challenges (e.g., applying DroidRA [Li et al., 2016b] to mitigate the impact of reflective calls on our static analysis approach.).

Third, the capability of our approach is limited by the number of Android APIs leveraged by real-world apps. Our approach cannot generate unit test cases for such APIs that have never been accessed by real-world Android apps. Nevertheless, we argue that this impact is not significant as the APIs that have never been used by app developers should have a low priority to be tested than the others that are frequently accessed. Compared to the latter case, the former ones will not cause problems such as crashes to Android apps. Subsequently, it will not impact the user experience and the reputation of the app developers.

Fourth, considering the generated parameter values are inferred from real-world apps, which might not reveal all possible semantic-related compatibility issues. In other words, the capability of our approach is limited by the values of parameters leveraged by real-world apps. As for our future work, we plan to integrate fuzzing techniques [Zhang et al., 2020, Patra and Pradel, 2016, Sui et al., 2011] into our approach so as to trigger as many compatibility issues as possible.

Fifth, our definition of compatibility issue is based on the observation that the same test case throws different errors or exceptions on different Android SDK versions. However, this might introduce false negatives because the tests that throw the same exception across all SDK versions are ignored. Nevertheless, we argue that this type of false-negative requires further human validation and thus cannot be determined automatically. In addition,

related works [Cai et al., 2019] also have not considered this situation.

Sixth, the tests generated by our approach may suffer from flaky tests. Indeed, non-deterministic return values may appear under different execution environments, leading to false positives. However, it is a non-trivial task to tackle flaky tests [Zolfaghari et al., 2021] because the root causes of flaky tests can be quite sophisticated. Nevertheless, as part of our future work, we plan to integrate other approaches developed by our fellow researchers to mitigate this long-standing challenge, e.g., by applying FlakeScanner [Dong et al., 2021] to mitigate the impact of flaky tests on our approach.

Seventh, the types of compatibility issues detected in our approach are the ones that are related to exceptions or return values. However, this will certainly not be complete to cover all possible cases. For example, the value of variables in the same API may evolve on different SDK versions. Indeed, as summarised by Liu et al. [Liu et al., 2022a], apart from compatibility issues raised by API signature/semantic changes and return value differences, there exist other types of compatibility issues, such as those introduced by field evolution, callback method changes, etc. Nevertheless, as also highlighted by Liu et al. [Liu et al., 2022a], the number of such compatibility issues is quite limited, suggesting that the impact of such cases on our approach may not be significant.

Eighth, the main objective of the generated test cases in this work is to pinpoint compatibility issues. The quality of these test cases (e.g., readability, overlaps, and maintenance) has hence not been considered. We therefore commit to further investigating the quality of the generated test cases in our future work.

Last but not least, our approach relies on existing code examples to generate test cases. However, the selected code examples may contain sub-optimal or erroneous API usages. Nevertheless, we argue that this impact is not significant as we extract code examples from real-world Android apps from Google Play, for which thousands of users might have used (hence tested) them in practice.

5.7 Related Work

Android API evolution is a critical issue in software maintenance [Nagappan and Shihab, 2016, Li et al., 2018a, Martin et al., 2016, Li et al., 2016a, Oliveira et al., 2018, Lamothe et al., 2020, Li et al., 2018c, Zhou and Walker, 2016, Dig and Johnson, 2005, Kapur et al., 2010, Liu et al., 2022b, Sun et al., 2021a]. McDonnell et al. [McDonnell et al., 2013] have shown that the Android system updates 115 APIs per month on average, while app developers usually adopt the new APIs much more slower. The slow adoption of API updates may raise various issues, such as security and compatibility. An empirical study on StackOverflow conducted by Linares-Vasquez et al. [Linares-Vásquez et al., 2014] suggests that API updates would trigger more discussions, especially if APIs are removed from the Android system. They also revealed that users are in more favour of apps that use less fault and change-prone APIs [Linares-Vásquez et al., 2013, Bavota et al., 2014], as these apps would likely introduce fewer failures, crashes and other bugs.

Android developers have long been suffered from compatibility issues due to the fast-evolving and fragmented nature of the Android ecosystem [Xia et al., 2020, Kamran et al., 2016, Li et al., 2017a, Nayebi et al., 2012, Zhao et al., 2022]. Researchers have proposed several solutions for detecting compatibility issues of Android APIs. Wei et al. [Wei et al., 2016, Wei et al., 2018] conducted an empirical study to investigate fragmentation-induced API compatibility issues and proposed a tool named FicFinder to detect such APIs. FicFinder identifies APIs with compatibility issues based on heuristic rules manually derived from a limited number of Android apps, which is expected to introduce high false negatives (i.e., missing undiscovered compatibility issues). Thereafter, several works have been proposed to leverage data-driven techniques that automatically mine compatibility issues from various sources such as Android code base and real-world apps. Li et al. proposed a tool named CiD to detect potential compatibility issues by mining the history of

the Android framework source code. CiD identifies Android APIs' lifetimes and finds if an app's declared supported versions conflict with its used APIs.

Comparable to our method, several works have been proposed to mine API usage from real-world apps. Scalabrino et al. [Scalabrino et al., 2019, Scalabrino et al., 2020] considered the APIs wrapped in a version check condition (e.g., *if (Build.VERSION.SDK_INT >= 21)*) to potentially have compatibility issues and developed a tool named ACRYL to extract such APIs from real-world apps. However, ACRYL can only detect APIs whose compatibility issue is already known by the developers (i.e., they are enclosed in the version check conditions by the developers), while our method is capable of detecting zero-day compatibility issues that the app developers are not yet aware of, or even Google itself. Other generic test generation tools, such as EvoSuite and Randoop[Pacheco and Ernst, 2007], are able to generate tests for Java classes. However, these tools do not directly aim at generating tests for APIs and they have been demonstrated as insufficient in pinpointing compatibility issues because of the lack of API usage knowledge.

5.8 Summary

In this work, we presented a novel prototype tool, JUnitTestGen, that mines existing Android API usages to generate API-focused unit test cases automatically for pinpointing potential compatibility issues caused by the fast evolution of the Android framework. Experimental results on thousands of real-world Android apps show that (1) JUnitTestGen is capable of automatically generating valid unit test cases for Android APIs with an 80.4% success rate; (2) the automatically generated test cases are useful for pinpointing API-induced compatibility issues, including not only signature-based but also semantics-based compatibility issues; and (3) JUnitTestGen outperforms the state-of-the-practice test generation tool, EvoSuite, and the state-of-the-art static compatibility detection tool, CiD, in pinpointing compatibility issues.

Chapter 6

Conclusions and Future Research

6.1 Key Contributions

This PhD project aimed to develop automatic tools that are tailored for detecting security and compatibility issues in Android applications. Particularly, I proposed three new approaches and prototype tools to detect these issues from three different aspects.

First, I proposed **a new static analysis-based approach, SEEKER**, which can detect privacy leaks originated from Android sensors in Android applications. Despite being needed to support the implementation of many diverse Android apps, mobile phone sensors can also be abused to achieve malicious behaviors. There have been many reports of apps that exploit sensors in Android devices to conduct malicious activities. For example, sensor data are known to be leaked out because they are not protected by any permissions in Android. There is hence a strong need to regulate the usage of mobile sensors so as to keep them from being exploited by malicious attackers. However, despite the fact that various efforts have been put in achieving this, i.e., detecting privacy leaks in Android apps, we have not yet found approaches to automatically detect sensor leaks in Android apps.

To address these issues, I proposed a novel tool, SEEKER [Sun et al., 2021a], for characterizing sensor leaks in Android apps. Specifically, I extended the famous FlowDroid [Arzt et al., 2014] tool to detect sensor-based data leaks in Android apps. SEEKER conducts sensor-focused static taint analyses directly on the Android apps' bytecode and reports not only sensor-triggered privacy leaks but also the sensor types involved in the leaks.

Our experimental results on a large scale of real-world Android apps indicate that our new SEEKER tool is effective in identifying all types of potential sensor leaks in Android apps. Our tool is not only capable of detecting sensor leaks, but also pinpointing general privacy leaks that are triggered by class fields. Although there are related works on sensor usage analysis, to the best of our knowledge, there is no other work that thoroughly analyses Android sensor leakage. Unlike previous works, our tool is the first one to characterize all kinds of sensor leaks in Android apps. I extended FlowDroid for supporting field sources detection (i.e., merged to FlowDroid via pull #385 on Github[Xiaoyu Sun, 2021]), which I believed could be adapted to analyze other sensitive field-triggered leaks. To benefit our fellow researchers and practitioners towards achieving this, I have made our approach open source at the following Github site.

The key contributions of this work are as follows:

(i) I have designed and implemented a prototype tool, SEEKER (Sensor leak finder), that leverages static analysis to automatically detect privacy leaks originated from Android sensors.

(ii) I applied SEEKER to analyze both malware and benign apps at a large scale. Our results show many sensor leaks that are overlooked by the state-of-the-art static analysis tool.

(iii) I have demonstrated the effectiveness of our tool by evaluating the sensor leaks it highlights.

Second, I proposed **a prototype program analysis tool, HISENDROID**, which performs a static code analysis to uncover hidden sensitive operations that will only be triggered under special circumstances such as at a specific location or in a certain time period. Additionally, HiSenDroid goes one step deeper to provide details aiming at helping security analysts understand why a given hidden sensitive operation is flagged as such. Experimental results over 20,000 apps, including both malicious and benign apps, show that hidden

sensitive operations are indeed quite frequently presented in Android apps and HiSenDroid is effective in automatically discovering them. Moreover, with the help of FlowDroid, a state-of-the-art static taint analyzer, I further experimentally find that hidden sensitive operations could eventually lead to privacy leaks.

The key contributions of this work are as follows:

(i) I designed and implemented a prototype tool HiSenDroid for analyzing hidden sensitive operations. I released HiSenDroid as an open source project [Sun, 2022] for supporting security analysts in their analysis needs and fostering further researches in this direction.

(ii) I evaluated HiSenDroid on a large-scale dataset that contains 10,000 benign and 10,000 malware samples, and discovered emerging anti-analysis techniques employed by malware samples, such as fulfilling certain restrictions related to *time*, *location*, *SMS message*, *system properties*, *package manager*, and other logics.

(iii) With the help of FlowDroid [Arzt et al., 2014], a static taint analyzer, I further experimentally showed that HSOs have been recurrently leveraged by attackers to leak sensitive user information.

Third, **I proposed a field-aware, inter-procedural backward data-flow analysis tool, JUNITTESTGEN**, which mines Android API usage to generate unit test cases for pinpointing compatibility issues. Despite being one of the largest and most popular projects, the official Android framework has only provided test cases for less than 30% of its APIs. Such a poor test case coverage rate has led to many compatibility issues that can cause apps to crash at runtime on specific Android devices, resulting in poor user experiences for both apps and the Android ecosystem. To mitigate this impact, various approaches have been proposed to automatically detect such compatibility issues. Unfortunately, these approaches have only focused on detecting signature-induced compatibility issues (i.e., a certain API does not exist in certain Android versions), leaving other equally important types of compatibility issues unresolved. One unresolved type is related to semantic

changes of APIs, which are non-trivial to pinpoint as it is yet not possible to comprehend automatically all the semantics of code. In this work, I proposed a novel prototype tool, JUnitTestGen, to fill this gap by mining existing Android API usage to generate unit test cases. After locating Android API usage in given real-world Android apps, JUnitTestGen performs inter-procedural backward data-flow analysis to generate a minimal executable code snippet (i.e., test case). Experimental results on thousands of real-world Android apps show that JUnitTestGen is effective in generating valid unit test cases for Android APIs. I showed that these generated test cases are indeed helpful for pinpointing compatibility issues, including ones involving semantic code changes.

In this work, I have made the following key contributions:

(i) I have designed and implemented a prototype tool JUnitTestGen that leverages a novel approach to automatically generate unit test cases for APIs based on their existing usages.

(ii) I have set up a reusable testing framework for pinpointing API-induced compatibility issues by automatically executing a large set of unit test cases on multiple Android devices.

(iii) I have demonstrated the effectiveness of JUnitTestGen by i) generating valid test cases for Android APIs and pinpointing problematic APIs that could induce compatibility issues if accessed by Android apps, ii) outperforming state-of-the-art tools on real-world apps in detecting a wider range of compatibility issues.

6.2 Future Work

My long-term research goal is to conduct comprehensive program analysis research in order to ensure better security and reliability of software systems. I am committed to solving difficult software problems using practical and elegant solutions.

The state-of-the-art static analysis techniques are not sufficient to find various defects

in real-world applications. This leaves many software defects undetected, causing serious problems for both users and developers. For example, attackers are reported to use complicated language features (e.g., reflection [Sun et al., 2021b], obfuscation [Dong et al., 2018c], and encryption [Götzfried and Müller, 2014]) to hide malicious operations. So I have the ambition to propose new approaches to do code unification to perform much more sound and comprehensive results. Specifically, I plan to unify native code, and javascript with the bytecode in Android applications to achieve this. After enhancing the static analysis techniques, I plan to further apply such techniques to help detect vulnerabilities or software defects which are overlooked before. This line of thinking provides for a number of opportunities for future research, incluidng but not limited to:

1. **React Native Code Unification:** Android code unification is needed to perform comprehensive static analysis for hybrid Android apps. In Android apps, dex bytecode cohabits with JavaScript code which can be used through the React Native Interface. React Native is an open-source UI software framework, which is well-known to be used for application development, including Android, Android TV, iOS, macOS, tvOS, Web and Windows [Rea, 2023]. Due to the challenge presented to analyze JavaScript code, it is most of the time overlooked by existing approaches. This limitation can be a severe threat to end users since malicious behaviour can be integrated in JavaScript code. Therefore, I have the ambition to propose a model unifying both the bytecode and the JavaScript code in Android apps. I plan to propose a first step toward this direction at the call-graph level and with more granularity at the statement level relying on heuristic-based defined statements.

2. **Typestate Misuse Detection in Android:** Android frameworks often provide complex functionality in order to support rich state transitions, which is a non-trivial task for programmers to understand and adopt correctly [Aldrich et al., 2009]. One essential aspect of correct API use is *typestate* [Mishra et al., 2016], which defines

legitimate sequences of operations that can be performed upon a given state type. A *typestate property* is a finite state machine [Fin, 2022], representing a sequence of behavioural type refinements such as "method A must be called after method B returns successfully". However, programmers are reported frequently ignore such disciplines [Krüger et al., 2019], causing Android defects and lost productivity when typestate APIs are misused in practice [Aldrich et al., 2009]. Even worse, applications that fail to follow correct state transitions (i.e., typestate misuses) may also be infected with vulnerabilities [Li et al., 2022b, Liu et al., 2016, Pan et al., 2020, Yan et al., 2017, Wang et al., 2020], including data corruption [Xu et al., 2015, Conti et al., 2015], privacy leaks [Lee et al., 2015, Serna, 2012] and hijacking attacks [Caballero et al., 2012, Fan et al., 2017].

To address this problem, I plan to design and implement a prototype tool to automatically detect typestate API misuse in real-world Android apps. Specifically, I am going to statically model the correct typestate usage based on the API comments in Android official documents, indicating the explicit disciplines of how typestate APIs should be performed. After that, I plan to perform intra- and inter-procedural data-flow analysis to extract the typestate API usage in real-world applications. After that, I will compare the typestate API usage with the correct typestate rules for pinpointing the violations. I believe that my tool can help app developers identify typestate misuse cases at the development stage to avoid these problems from being distributed into the community.

3. **Dynamic Software Testing:** The current state-of-the-art works targeting software defects detection leverage static analysis techniques to achieve their purpose. However, all software detects cannot be statically detected under every circumstance, introducing false positives and false negatives. To mitigate this problem, I plan to

leverage dynamic program testing techniques to perform defects analysis as dynamic malware analysis methods generally provide better precision than purely static methods.

To trigger malicious behaviours, I plan to develop a tool for automatically generating inputs that will trigger certain sensitive APIs (e.g., the APIs are protected by permission) in the source code. By dynamically triggering the target APIs, I am able to determine whether there exist malicious operations. In addition, I will then integrate my tool with state-of-the-art static taint analysis tools (e.g., FlowDroid [Arzt et al., 2014]) to offer better precision.

4. **Enhance Program Analysis Techniques:** Since the state-of-the-art static analysis techniques are not sufficient to find various defects in real-world applications. This would leave a bunch of defects undetected, causing serious problems for both users and developers. For example, many flaws have been found in traditional static analysis tools [Ami et al., 2021, Bonett et al., 2018], including missing callbacks, missing implicit call, incorrect modeling of anonymous classes and incorrect modeling of asynchronous methods, etc. These flaws can be exploited by attackers to hide malicious activities from code analyzers. To this end, much research efforts should be put into improving state-of-the-art static analysis techniques by making such tools more sound and effective.

   To address this problem, I plan to first systematically identify the features of the existing flaws and summrize the reasons why the traditional analyzers fail in such scenarios. In addition, given the detected code features, prototype tools can be developed to automatically model such sophisticated code types to help enhance the state-of-the-art static program analysis techniques. I believe that our enhanced tech-

niques can be used to detect more security issues in the wild.

6.3 Summary

In this Ph.D. thesis I focused on detecting security and compatibility issues in Android applications. I developed three tools, i.e., SEEKER, HISENDROID, JUNITTESTGEN for improving the traditional program analysis techniques in detecting wider range of security and compatibility issues. Specifically, SEEKER helps extend FlowDroid for supporting field sources detection, HiSenDroid reveals hidden sensitive operations which have been overlooked by state-of-the-art tools and JUnitTestGen is an effective tool to detect compatibility issues through a dynamic testing approach that fulfills its objective by automatically generating valid test cases. I plan to enhance the soundness and comprehensiveness of existing program analysis techniques for identifying wider range of security and compatibility issues in our future work.

References

[tam,] Android security: Adding tampering detection to your app. https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app#4-1-emulator. Last updated: 2021-11-20.

[And,] Android.hehe: Malware now disconnects phone calls. https://www.fireeye.com/blog/threat-research/2014/01/android-hehemalware-now-disconnects-phone-calls.html. Last updated: 2021-11-20.

[Flo,] Flowdroid development branch. https://github.com/secure-software-engineering/FlowDroid/tree/develop. Last updated: 2021-10-14.

[Hac,] Hacking team rcsandroid spying tool listens to calls; roots devices to get in. https://www.trendmicro.com/en_us/research/15/g/hacking-team-rcsandroid-spying-tool-listens-to-calls-roots-devices-html. Last updated: 2021-11-20.

[Vir, 2020] (2020). *Virusshare*.

[Vir, 2021] (2021). *VirusTotal*.

[Fin, 2022] (2022). Finite-state machine. https://en.wikipedia.org/wiki/Finite-state_machine. Last updated: 2022-07-13.

[Rea, 2023] (2023). React native. https://en.wikipedia.org/wiki/React_Native. Last updated: 2023-05-07.

[Kex, 2022] (24 April 2022). Kex. https://github.com/vorpal-research/kex/tree/sbst-contest. Online; accessed 24 April 2022.

[Aafer et al., 2018] Aafer, Y., Tao, G., Huang, J., Zhang, X., and Li, N. (2018). Precise android api protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1151–1164.

[Al-Haiqi et al., 2013] Al-Haiqi, A., Ismail, M., and Nordin, R. (2013). Keystrokes inference attack on android: A comparative evaluation of sensors and their fusion. *Journal of ICT Research & Applications*, 7(2).

[Aldrich et al., 2009] Aldrich, J., Sunshine, J., Saini, D., and Sparks, Z. (2009). Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022.

[Allen et al., 2006] Allen, F. R., Ambikairajah, E., Lovell, N. H., and Celler, B. G. (2006). Classification of a known sequence of motions and postures from accelerometry data using adapted gaussian mixture models. *Physiological measurement*, 27(10):935.

[Allix et al., 2016] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016). Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE.

[Ami et al., 2022] Ami, A. S., Cooper, N., Kafle, K., Moran, K., Poshyvanyk, D., and Nadkarni, A. (2022). Why crypto-detectors fail: A systematic evaluation of cryptographic

misuse detection techniques. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 614–631. IEEE.

[Ami et al., 2021] Ami, A. S., Kafle, K., Moran, K., Nadkarni, A., and Poshyvanyk, D. (2021). Systematic mutation-based evaluation of the soundness of security-focused android static analysis techniques. *ACM Transactions on Privacy and Security (TOPS)*, 24(3):1–37.

[Anthony Scarsella, 2020] Anthony Scarsella, W. S. (2020). Worldwide smartphone market shares, 2019. `https://www.idc.com/getdoc.jsp?containerId=US46194820`.

[Aonzo et al., 2020] Aonzo, S., Georgiu, G. C., Verderame, L., and Merlo, A. (2020). Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403.

[Arp et al., 2014] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., and Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26.

[Arzt et al., 2013] Arzt, S., Rasthofer, S., and Bodden, E. (2013). Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*.

[Arzt et al., 2014] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269.

[Au et al., 2012] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. (2012). Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228.

[Avdiienko et al., 2015] Avdiienko, V., Kuznetsov, K., Gorla, A., Zeller, A., Arzt, S., Rasthofer, S., and Bodden, E. (2015). Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 426–436. IEEE.

[Aviv et al., 2012] Aviv, A. J., Sapp, B., Blaze, M., and Smith, J. M. (2012). Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th annual computer security applications conference*, pages 41–50.

[Ba et al., 2020] Ba, Z., Zheng, T., Zhang, X., Qin, Z., Li, B., Liu, X., and Ren, K. (2020). Learning-based practical smartphone eavesdropping with built-in accelerometer. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium*, pages 23–26.

[Backes et al., 2016] Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., and Weisgerber, S. (2016). On demystifying the android application framework:{Re-Visiting} android permission specification analysis. In *25th USENIX security symposium (USENIX security 16)*, pages 1101–1118.

[Balzarotti et al., 2010] Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., and Vigna, G. (2010). Efficient detection of split personalities in malware. In *NDSS*. Citeseer.

[Bavota et al., 2014] Bavota, G., Linares-Vasquez, M., Bernal-Cardenas, C. E., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2014). The impact of api change-and fault-

proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407.

[Biedermann et al., 2015] Biedermann, S., Katzenbeisser, S., and Szefer, J. (2015). Hard drive side-channel attacks using smartphone magnetic field sensors. In *International Conference on Financial Cryptography and Data Security*, pages 489–496. Springer.

[Block and Noubir, 2018] Block, K. and Noubir, G. (2018). My magnetometer is telling you where i've been? a mobile device permissionless location attack. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 260–270.

[Bo et al., 2019] Bo, L., Fengjun, L., Guanghui, W., et al. (2019). *I Know What You Type on Your Phone: Keystroke Inference on Android Device Using Deep Learning*. PhD thesis, University of Kansas.

[Bodden, 2012] Bodden, E. (2012). Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 3–8.

[Bonett et al., 2018] Bonett, R., Kafle, K., Moran, K., Nadkarni, A., and Poshyvanyk, D. (2018). Discovering flaws in security-focused static analysis tools for android using systematic mutation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1263–1280.

[Brumley et al., 2008] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., and Yin, H. (2008). Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer.

[Caballero et al., 2012] Caballero, J., Grieco, G., Marron, M., and Nappa, A. (2012). Undangle: early detection of dangling pointers in use-after-free and double-free vulnera-

bilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143.

[Cai et al., 2019] Cai, H., Zhang, Z., Li, L., and Fu, X. (2019). A large-scale study of application incompatibilities in android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 216–227.

[Cai and Chen, 2011] Cai, L. and Chen, H. (2011). Touchlogger: Inferring keystrokes on touch screen from smartphone motion. *HotSec*, 11(2011):9.

[Cai and Chen, 2012] Cai, L. and Chen, H. (2012). On the practicality of motion based keystroke inference attack. In *International Conference on Trust and Trustworthy Computing*, pages 273–290. Springer.

[Cai et al., 2009] Cai, L., Machiraju, S., and Chen, H. (2009). Defending against sensor-sniffing attacks on mobile phones. In *Proceedings of the 1st ACM workshop on Networking, systems, and applications for mobile handhelds*, pages 31–36.

[Chakraborty et al., 2017] Chakraborty, S., Ouyang, W., and Srivastava, M. (2017). Lightspy: Optical eavesdropping on displays using light sensors on mobile devices. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2980–2989. IEEE.

[Chang et al., 2009] Chang, K.-h., Hightower, J., and Kveton, B. (2009). Inferring identity using accelerometers in television remote controls. In *International Conference on Pervasive Computing*, pages 151–167. Springer.

[Chen et al., 2008] Chen, X., Andersen, J., Mao, Z. M., Bailey, M., and Nazario, J. (2008). Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN)*, pages 177–186. IEEE.

[Chen et al., 2021] Chen, X., Chen, W., Liu, K., Chen, C., and Li, L. (2021). A comparative study of smartphone and smartwatch apps. In *The 2021 ACM/SIGAPP Symposium on Applied Computing (SAC 2021)*.

[Conti et al., 2015] Conti, M., Crane, S., Davi, L., Franz, M., Larsen, P., Negro, M., Liebchen, C., Qunaibit, M., and Sadeghi, A.-R. (2015). Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963.

[Costamagna et al., 2018] Costamagna, V., Zheng, C., and Huang, H. (2018). Identifying and evading android sandbox through usage-profile based fingerprints. In *Proceedings of the First Workshop on Radical and Experiential Security*, pages 17–23.

[Cova et al., 2010] Cova, M., Kruegel, C., and Vigna, G. (2010). Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290.

[Crandall et al., 2006] Crandall, J. R., Wassermann, G., de Oliveira, D. A., Su, Z., Wu, S. F., and Chong, F. T. (2006). Temporal search: Detecting hidden malware timebombs with virtual machines. *ACM SIGOPS Operating Systems Review*, 40(5):25–36.

[Diao et al., 2016] Diao, W., Liu, X., Li, Z., and Zhang, K. (2016). Evading android runtime analysis through detecting programmed interactions. In *Proceedings of the 9th ACM conference on security & privacy in wireless and mobile networks*, pages 159–164.

[Dig and Johnson, 2005] Dig, D. and Johnson, R. (2005). The role of refactorings in api evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398. IEEE.

[Dong et al., 2018a] Dong, F., Wang, H., Li, L., Guo, Y., Bissyandé, T. F., Liu, T., Xu, G., and Klein, J. (2018a). Frauddroid: Automated ad fraud detection for android apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 257–268.

[Dong et al., 2018b] Dong, F., Wang, H., Li, L., Guo, Y., Xu, G., and Zhang, S. (2018b). How do mobile apps violate the behavioral policy of advertisement libraries? In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, pages 75–80.

[Dong et al., 2018c] Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., Xu, F., Chen, K., Wang, X., and Zhang, K. (2018c). Understanding android obfuscation techniques: A large-scale investigation in the wild. In *Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I*, pages 172–192. Springer.

[Dong et al., 2021] Dong, Z., Tiwari, A., Yu, X. L., and Roychoudhury, A. (2021). Flaky test detection in android via event order exploration. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 367–378.

[Egele et al., 2008] Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2008). A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):1–42.

[Enck et al., 2014] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: an information-flow track-

ing system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29.

[Fan et al., 2017] Fan, X., Sui, Y., Liao, X., and Xue, J. (2017). Boosting the precision of virtual call integrity protection with partial pointer analysis for c++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340.

[Fay and Proschan, 2010] Fay, M. P. and Proschan, M. A. (2010). Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys*, 4:1.

[Fraser and Arcuri, 2011a] Fraser, G. and Arcuri, A. (2011a). Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*, pages 31–40. IEEE.

[Fraser and Arcuri, 2011b] Fraser, G. and Arcuri, A. (2011b). Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419.

[Fratantonio et al., 2016] Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., and Vigna, G. (2016). Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*, pages 377–396. IEEE.

[Gao et al., 2020] Gao, J., Li, L., Kong, P., Bissyandé, T. F., and Klein, J. (2020). Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation. In *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*.

[Giallanza et al., 2019] Giallanza, T., Siems, T., Smith, E., Gabrielsen, E., Johnson, I., Thornton, M. A., and Larson, E. C. (2019). Keyboard snooping from mobile phone arrays with mixed convolutional and recurrent neural networks. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(2):1–22.

[Glanz et al., 2020] Glanz, L., Müller, P., Baumgärtner, L., Reif, M., Amann, S., Anthonysamy, P., and Mezini, M. (2020). Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 694–707.

[Google, 2021a] Google (2021a). LauncherApps. `https://developer.android.com/reference/android/content/pm/LauncherApps#hasShortcutHostPermission()`. Online; accessed 27 January 2021.

[Google, 2021b] Google (2021b). Sensors Overview. `https://developer.android.com/guide/topics/sensors/sensors_overview.html`. Online; accessed 12 March 2021.

[Google, 2022] Google (2022). The google android security team's classifications for potentially harmful applications. `https://developers.google.com/android/play-protect/phacategories`. Online; accessed October 2022.

[Google, 2022] Google (29 Nov 2022). How Android protects you from scams and phishing attacks. `https://blog.google/products/android/android-scam-and-phishing-protection/`. Online; accessed 29 Nov 2022.

[Gordon et al., 2015] Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., and Rinard, M. C. (2015). Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110.

[Götzfried and Müller, 2014] Götzfried, J. and Müller, T. (2014). Analysing android's full disk encryption feature. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 5(1):84–100.

[Ham and Park, 2011] Ham, H. K. and Park, Y. B. (2011). Mobile application compatibility test system design for android fragmentation. In *International Conference on Advanced Software Engineering and Its Applications*, pages 314–320. Springer.

[Han et al., 2012] Han, D., Zhang, C., Fan, X., Hindle, A., Wong, K., and Stroulia, E. (2012). Understanding android fragmentation with topic analysis of vendor-specific bugs. In *2012 19th Working Conference on Reverse Engineering*, pages 83–92. IEEE.

[He et al., 2018] He, D., Li, L., Wang, L., Zheng, H., Li, G., and Xue, J. (2018). Understanding and detecting evolution-induced compatibility issues in android apps. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 167–177. IEEE.

[Hodges and Buckley, 2018] Hodges, D. and Buckley, O. (2018). Reconstructing what you said: Text inference using smartphone motion. *IEEE Transactions on Mobile Computing*, 18(4):947–959.

[Holmquist et al., 2001] Holmquist, L. E., Mattern, F., Schiele, B., Alahuhta, P., Beigl, M., and Gellersen, H.-W. (2001). Smart-its friends: A technique for users to easily establish connections between smart artefacts. In *international conference on Ubiquitous Computing*, pages 116–122. Springer.

[Huang et al., 2018] Huang, H., Wei, L., Liu, Y., and Cheung, S.-C. (2018). Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 532–542.

[Huang et al., 2019] Huang, Y., Guan, X., Chen, H., Liang, Y., Yuan, S., and Ohtsuki, T. (2019). Risk assessment of private information inference for motion sensor embedded iot devices. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 4(3):265–275.

[Jiang and Su, 2009] Jiang, L. and Su, Z. (2009). Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92.

[Jing et al., 2014] Jing, Y., Zhao, Z., Ahn, G.-J., and Hu, H. (2014). Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225.

[Kamran et al., 2016] Kamran, M., Rashid, J., and Nisar, M. W. (2016). Android fragmentation classification, causes, problems and solutions. *International Journal of Computer Science and Information Security*, 14(9):992.

[Kapur et al., 2010] Kapur, P., Cossette, B., and Walker, R. J. (2010). Refactoring references for library migration. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 726–738.

[Kaspersky-Lab, 2022] Kaspersky-Lab (29 Nov 2022). Mobile Security: Android vs iOS — which one is safer? `https://www.kaspersky.com/resource-center/threats/android-vs-iphone-mobile-security`. Online; accessed 29 Nov 2022.

[Kechagia et al., 2019] Kechagia, M., Devroey, X., Panichella, A., Gousios, G., and van Deursen, A. (2019). Effective and efficient api misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 192–203.

[Kirat and Vigna, 2015] Kirat, D. and Vigna, G. (2015). Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 769–780.

[Kirat et al., 2014] Kirat, D., Vigna, G., and Kruegel, C. (2014). Barecloud: bare-metal analysis-based evasive malware detection. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 287–301.

[Klieber et al., 2014] Klieber, W., Flynn, L., Bhosale, A., Jia, L., and Bauer, L. (2014). Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6.

[Kong et al., 2018] Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., and Klein, J. (2018). Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*.

[Krüger et al., 2019] Krüger, S., Späth, J., Ali, K., Bodden, E., and Mezini, M. (2019). Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, 47(11):2382–2400.

[Lam et al., 2011] Lam, P., Bodden, E., Lhoták, O., and Hendren, L. (2011). The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35.

[Lamothe et al., 2020] Lamothe, M., Shang, W., and Chen, T.-H. P. (2020). A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering*.

[Lee et al., 2015] Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., and Lee, W. (2015). Preventing use-after-free with dangling pointers nullification. In *NDSS*.

[Lee and Lee, 2015] Lee, W.-H. and Lee, R. B. (2015). Multi-sensor authentication to improve smartphone security. In *2015 International conference on information systems security and privacy (ICISSP)*, pages 1–11. IEEE.

[Lester et al., 2004] Lester, J., Hannaford, B., and Borriello, G. (2004). "are you with me?"–using accelerometers to determine if two devices are carried by the same person. In *International Conference on Pervasive Computing*, pages 33–50. Springer.

[Lhoták and Hendren, 2003] Lhoták, O. and Hendren, L. (2003). Scaling java points-to analysis using s park. In *International Conference on Compiler Construction*, pages 153–169. Springer.

[Li et al., 2022a] Li, C., Chen, X., Sun, R., Xue, J., Wen, S., Ahmed, M. E., Camtepe, S., and Xiang, Y. (2022a). Cross-language android permission specification. In *Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[Li et al., 2015a] Li, L., Allix, K., Li, D., Bartel, A., Bissyandé, T. F., and Klein, J. (2015a). Potential component leaks in android apps: An investigation into a new feature set for malware detection. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 195–200. IEEE.

[Li et al., 2015b] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2015b). ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *Proceedings of the 30th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC 2015)*.

[Li et al., 2015c] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. (2015c). Iccta: Detecting inter-

component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE.

[Li et al., 2018a] Li, L., Bissyandé, T. F., and Klein, J. (2018a). Moonlightbox: Mining android api histories for uncovering release-time inconsistencies. In *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*.

[Li et al., 2019] Li, L., Bissyandé, T. F., and Klein, J. (2019). Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering (TSE)*.

[Li et al., 2016a] Li, L., Bissyandé, T. F., Le Traon, Y., and Klein, J. (2016a). Accessing inaccessible android apis: An empirical study. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–422. IEEE.

[Li et al., 2016b] Li, L., Bissyandé, T. F., Octeau, D., and Klein, J. (2016b). Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 318–329.

[Li et al., 2017a] Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Le Traon, Y. (2017a). Static analysis of android apps: A systematic literature review. *Information and Software Technology*.

[Li et al., 2018b] Li, L., Bissyandé, T. F., Wang, H., and Klein, J. (2018b). Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–163.

[Li et al., 2018c] Li, L., Gao, J., Bissyandé, T. F., Ma, L., Xia, X., and Klein, J. (2018c). Characterising deprecated android apis. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 254–264.

[Li et al., 2017b] Li, L., Li, D., Bissyandé, T. F., Klein, J., Le Traon, Y., Lo, D., and Cavallaro, L. (2017b). Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security (TIFS)*.

[Li et al., 2022b] Li, T., Bai, J.-J., Sui, Y., and Hu, S.-M. (2022b). Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–872.

[Lin et al., 2012] Lin, C.-C., Chang, C.-C., Liang, D., and Yang, C.-H. (2012). A new non-intrusive authentication method based on the orientation sensor for smartphone users. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 245–252. IEEE.

[Lin and Seibel, 2019] Lin, J. and Seibel, J. (2019). Motion-based side-channel attack on mobile keystrokes.

[Linares-Vásquez et al., 2013] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2013). Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487.

[Linares-Vásquez et al., 2014] Linares-Vásquez, M., Bavota, G., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2014). How do api changes trigger stack overflow discussions? a study on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*, pages 83–94.

[Lindorfer et al., 2011] Lindorfer, M., Kolbitsch, C., and Comparetti, P. M. (2011). Detecting environment-sensitive malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 338–357. Springer.

[Ling et al., 2019] Ling, Z., Li, Z., Chen, C., Luo, J., Yu, W., and Fu, X. (2019). I know what you enter on gear vr. In *2019 IEEE Conference on Communications and Network Security (CNS)*, pages 241–249. IEEE.

[Liu et al., 2009] Liu, J., Zhong, L., Wickramasuriya, J., and Vasudevan, V. (2009). uwave: Accelerometer-based personalized gesture recognition and its applications. *Pervasive and Mobile Computing*, 5(6):657–675.

[Liu et al., 2021a] Liu, P., Li, L., Yan, Y., Fazzini, M., and Grundy, J. (2021a). Identifying and characterizing silently-evolved methods in the android api. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2021)*, pages 308–317. IEEE.

[Liu et al., 2020a] Liu, P., Li, L., Zhao, Y., Sun, X., and Grundy, J. (2020a). Androzooopen: Collecting large-scale open source android apps for the research community. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 548–552.

[Liu et al., 2022a] Liu, P., Zhao, Y., Cai, H., Fazzini, M., Grundy, J., and Li, L. (2022a). Automatically detecting api-induced compatibility issues in android apps: A comparative analysis (replicability study). *arXiv preprint arXiv:2205.15561*.

[Liu et al., 2020b] Liu, T., Wang, H., Li, L., Luo, X., Dong, F., Guo, Y., Wang, L., Bissyandé, T. F., and Klein, J. (2020b). Maddroid: Characterising and detecting devious ad content for android apps. In *The Web Conference 2020 (WWW 2020)*.

[Liu et al., 2015a] Liu, X., Liu, J., and Wang, W. (2015a). Exploring sensor usage behaviors of android applications based on data flow analysis. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE.

[Liu et al., 2018] Liu, X., Liu, J., Wang, W., He, Y., and Zhang, X. (2018). Discovering and understanding android sensor usage behaviors with data flow analysis. *World Wide Web*, 21(1):105–126.

[Liu et al., 2015b] Liu, X., Zhou, Z., Diao, W., Li, Z., and Zhang, K. (2015b). When good becomes evil: Keystroke inference with smartwatch. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1273–1285.

[Liu et al., 2021b] Liu, Y., Li, L., Kong, P., Sun, X., and Bissyandé, T. F. (2021b). A first look at security risks of android tv apps. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 59–64. IEEE.

[Liu and Li, 2019] Liu, Y. and Li, Z. (2019). aleak: Context-free side-channel from your smart watch leaks your typing privacy. *IEEE Transactions on Mobile Computing*, 19(8):1775–1788.

[Liu et al., 2021c] Liu, Y., Tantithamthavorn, C., Li, L., and Liu, Y. (2021c). Deep learning for android malware defenses: a systematic literature review. *arXiv preprint arXiv:2103.05292*.

[Liu et al., 2022b] Liu, Y., Tantithamthavorn, C., Li, L., and Liu, Y. (2022b). Deep learning for android malware defenses: a systematic literature review. *ACM Computing Surveys (CSUR)*.

[Liu et al., 2022c] Liu, Y., Tantithamthavorn, C., Li, L., and Liu, Y. (2022c). Explainable ai for android malware detection: Towards understanding why the models perform so

well? In *The 33rd IEEE International Symposium on Software Reliability Engineering (ISSRE 2022)*.

[Liu et al., 2014] Liu, Y., Xu, C., and Cheung, S.-C. (2014). Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*, pages 1013–1024.

[Liu et al., 2016] Liu, Y., Xu, C., Cheung, S.-C., and Terragni, V. (2016). Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 396–409.

[Livshits et al., 2015] Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. (2015). In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46.

[Lu et al., 2018] Lu, C. X., Du, B., Wen, H., Wang, S., Markham, A., Martinovic, I., Shen, Y., and Trigoni, N. (2018). Snoopy: Sniffing your smartwatch passwords via deep sequence learning. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(4):1–29.

[Maiti et al., 2018a] Maiti, A., Heard, R., Sabra, M., and Jadliwala, M. (2018a). Towards inferring mechanical lock combinations using wrist-wearables as a side-channel. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 111–122.

[Maiti et al., 2015] Maiti, A., Jadliwala, M., He, J., and Bilogrevic, I. (2015). (smart) watch your taps: Side-channel keystroke inference attacks using smartwatches. In *Pro-*

ceedings of the 2015 ACM International Symposium on Wearable Computers, pages 27–30.

[Maiti et al., 2018b] Maiti, A., Jadliwala, M., He, J., and Bilogrevic, I. (2018b). Side-channel inference attacks on mobile keypads using smartwatches. *IEEE Transactions on Mobile Computing*, 17(9):2180–2194.

[Mariconti et al., 2016] Mariconti, E., Onwuzurike, L., Andriotis, P., De Cristofaro, E., Ross, G., and Stringhini, G. (2016). Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433*.

[Marquardt et al., 2011] Marquardt, P., Verma, A., Carter, H., and Traynor, P. (2011). (sp) iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 551–562.

[Martin et al., 2016] Martin, W., Sarro, F., Jia, Y., Zhang, Y., and Harman, M. (2016). A survey of app store analysis for software engineering. *IEEE transactions on software engineering*, 43(9):817–847.

[McDonnell et al., 2013] McDonnell, T., Ray, B., and Kim, M. (2013). An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79. IEEE.

[Mehrnezhad et al., 2018] Mehrnezhad, M., Toreini, E., Shahandashti, S. F., and Hao, F. (2018). Stealing pins via mobile sensors: actual risk versus user perception. *International Journal of Information Security*, 17(3):291–313.

[Michalevsky et al., 2014] Michalevsky, Y., Boneh, D., and Nakibly, G. (2014). Gyrophone: Recognizing speech from gyroscope signals. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 1053–1067.

[Miluzzo et al., 2012] Miluzzo, E., Varshavsky, A., Balakrishnan, S., and Choudhury, R. R. (2012). Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 323–336.

[Mishra et al., 2016] Mishra, A., Kanade, A., and Srikant, Y. (2016). Asynchrony-aware static analysis of android applications. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 163–172. IEEE.

[Moser et al., 2007] Moser, A., Kruegel, C., and Kirda, E. (2007). Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE.

[Mutchler et al., 2016] Mutchler, P., Safaei, Y., Doupé, A., and Mitchell, J. (2016). Target fragmentation in android apps. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 204–213. IEEE.

[Nagappan and Shihab, 2016] Nagappan, M. and Shihab, E. (2016). Future trends in software engineering research for mobile apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 21–32. IEEE.

[Narain et al., 2014] Narain, S., Sanatinia, A., and Noubir, G. (2014). Single-stroke language-agnostic keylogging using stereo-microphones and domain specific machine learning. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 201–212.

[Nayebi et al., 2012] Nayebi, F., Desharnais, J.-M., and Abran, A. (2012). The state of the art of mobile application usability evaluation. In *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–4. IEEE.

[Nguyen, 2015] Nguyen, T. (2015). Using unrestricted mobile sensors to infer tapped and traced user inputs. In *2015 12th International Conference on Information Technology-New Generations*, pages 151–156. IEEE.

[Norboev et al., 2017] Norboev, R., Hossain, Z., Luo, L., and Zeng, Q. (2017). On the robustness of stochastic stealthy network against android app repackaging. Technical report, Technical Report. Temple University.

[Oberheide and Miller, 2012] Oberheide, J. and Miller, C. (2012). Dissecting the android bouncer. *SummerCon2012, New York*, 95:110.

[Octeau et al., 2016] Octeau, D., Jha, S., Dering, M., Mcdaniel, P., Bartel, A., Li, L., Klein, J., and Le Traon, Y. (2016). Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL 2016)*.

[Octeau et al., 2015] Octeau, D., Luchaup, D., Dering, M., Jha, S., and McDaniel, P. (2015). Composite constant propagation: Application to android inter-component communication analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 77–88. IEEE.

[Octeau et al., 2013] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Le Traon, Y. (2013). Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 543–558.

[Oliveira et al., 2018] Oliveira, J., Borges, D., Silva, T., Cacho, N., and Castor, F. (2018). Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software*, 136:1–18.

[Owusu et al., 2012] Owusu, E., Han, J., Das, S., Perrig, A., and Zhang, J. (2012). Accessory: password inference using accelerometers on smartphones. In *proceedings of the twelfth workshop on mobile computing systems & applications*, pages 1–6.

[Pacheco and Ernst, 2007] Pacheco, C. and Ernst, M. D. (2007). Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816.

[Pan et al., 2020] Pan, L., Cui, B., Liu, H., Yan, J., Wang, S., Yan, J., and Zhang, J. (2020). Static asynchronous component misuse detection for android applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 952–963.

[Pan et al., 2017] Pan, X., Wang, X., Duan, Y., Wang, X., and Yin, H. (2017). Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps. In *NDSS*.

[Papp et al., 2019] Papp, D., Tarrach, T., and Buttyán, L. (2019). Towards detecting trigger-based behavior in binaries: Uncovering the correct environment. In *International Conference on Software Engineering and Formal Methods*, pages 491–509. Springer.

[Patra and Pradel, 2016] Patra, J. and Pradel, M. (2016). Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*.

[Petsas et al., 2014] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., and Ioannidis, S. (2014). Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, pages 1–6.

[Pirttikangas et al., 2006] Pirttikangas, S., Fujinami, K., and Nakajima, T. (2006). Feature selection and activity recognition from wearable sensors. In *International symposium on ubiquitious computing systems*, pages 516–527. Springer.

[Qiu et al., 2018] Qiu, L., Wang, Y., and Rubin, J. (2018). Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 176–186, New York, NY, USA. Association for Computing Machinery.

[Rasthofer et al., 2016] Rasthofer, S., Arzt, S., Miltenberger, M., and Bodden, E. (2016). Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*.

[Rasthofer et al., 2017] Rasthofer, S., Arzt, S., Triller, S., and Pradel, M. (2017). Making malory behave maliciously: Targeted fuzzing of android execution environments. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 300–311. IEEE.

[Ravi et al., 2005] Ravi, N., Dandekar, N., Mysore, P., and Littman, M. L. (2005). Activity recognition from accelerometer data. In *Aaai*, volume 5, pages 1541–1546. Pittsburgh, PA.

[Ren et al., 2020] Ren, X., Ye, X., Xing, Z., Xia, X., Xu, X., Zhu, L., and Sun, J. (2020). Api-misuse detection driven by fine-grained api-constraint knowledge graph. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 461–472. IEEE.

[Samhi and Bartel, 2021] Samhi, J. and Bartel, A. (2021). On the (in) effectiveness of static logic bomb detector for android apps. *IEEE Transactions on Dependable and Secure Computing*.

[Samhi et al., 2021] Samhi, J., Bartel, A., Bissyandé, T. F., and Klein, J. (2021). Raicc: Revealing atypical inter-component communication in android apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1398–1409. IEEE.

[Samhi et al., 2022] Samhi, J., Gao, J., Daoudi, N., Graux, P., Hoyez, H., Sun, X., Allix, K., Bissyandé, T. F., and Klein, J. (2022). Jucify: a step towards android code unification for enhanced static analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1232–1244. IEEE.

[Sarkisyan et al., 2015] Sarkisyan, A., Debbiny, R., and Nahapetian, A. (2015). Wristsnoop: Smartphone pins prediction using smartwatch motion sensors. In *2015 IEEE international workshop on information forensics and security (WIFS)*, pages 1–6. IEEE.

[Scalabrino et al., 2019] Scalabrino, S., Bavota, G., Linares-Vásquez, M., Lanza, M., and Oliveto, R. (2019). Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 288–298. IEEE.

[Scalabrino et al., 2020] Scalabrino, S., Bavota, G., Linares-Vásquez, M., Piantadosi, V., Lanza, M., and Oliveto, R. (2020). Api compatibility issues in android: Causes and effectiveness of data-driven detection techniques. *Empirical Software Engineering*, 25(6):5006–5046.

[Schlegel et al., 2011] Schlegel, R., Zhang, K., Zhou, X.-y., Intwala, M., Kapadia, A., and Wang, X. (2011). Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33.

[Serna, 2012] Serna, F. J. (2012). The info leak era on software exploitation. *Black Hat USA*.

[Shen et al., 2015] Shen, C., Pei, S., Yang, Z., and Guan, X. (2015). Input extraction via motion-sensor behavior analysis on smartphones. *Computers & Security*, 53:143–155.

[Sikder et al., 2021] Sikder, A. K., Petracca, G., Aksu, H., Jaeger, T., and Uluagac, A. S. (2021). A survey on sensor-based threats and attacks to smart devices and applications. *IEEE Communications Surveys & Tutorials*.

[Song et al., 2016] Song, C., Lin, F., Ba, Z., Ren, K., Zhou, C., and Xu, W. (2016). My smartphone knows what you print: Exploring smartphone-based side-channel attacks against 3d printers. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 895–907.

[Stack OverFlow, 2021] Stack OverFlow (2021). Permissions needed for NotificationManager. `https://stackoverflow.com/questions/41308512/permissions-needed-for-notificationmanager`. Online; accessed 27 January 2021.

[Sui et al., 2011] Sui, A.-F., Tang, W., Hu, J. J., and Li, M. Z. (2011). An effective fuzz input generation method for protocol testing. In *2011 IEEE 13th International Conference on Communication Technology*, pages 728–731. IEEE.

[Sun, 2022] Sun, X. (30 Nov 2022). Hisendroid. `https://bitbucket.org/se_anonymous/workspace/projects/HIS`. Last updated: 2021-03-30.

[Sun et al., 2021a] Sun, X., Chen, X., Liu, K., Wen, S., Li, L., and Grundy, J. (2021a). Characterizing sensor leaks in android apps. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 498–509. IEEE.

[Sun et al., 2022] Sun, X., Chen, X., Zhao, Y., Liu, P., Grundy, J., and Li, L. (2022). Mining android api usage to generate unit test cases for pinpointing compatibility issues. *arXiv preprint arXiv:2208.13417*.

[Sun et al., 2021b] Sun, X., Li, L., Bissyandé, T. F., Klein, J., Octeau, D., and Grundy, J. (2021b). Taming reflection: An essential step toward whole-program analysis of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–36.

[Vallée-Rai et al., 2010] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (2010). Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224.

[Vidas and Christin, 2014] Vidas, T. and Christin, N. (2014). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458.

[Vogl et al., 2021] Vogl, S., Schweikl, S., Fraser, G., Arcuri, A., Campos, J., and Panichella, A. (2021). Evosuite at the sbst 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 28–29. IEEE.

[Vuagnoux and Pasini, 2009] Vuagnoux, M. and Pasini, S. (2009). Compromising electromagnetic emanations of wired and wireless keyboards. In *USENIX security symposium*, pages 1–16.

[Wang et al., 2015] Wang, H., Lai, T. T.-T., and Roy Choudhury, R. (2015). Mole: Motion leaks through smartwatch sensors. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 155–166.

[Wang et al., 2020] Wang, H., Xie, X., Li, Y., Wen, C., Li, Y., Liu, Y., Qin, S., Chen, H., and Sui, Y. (2020). Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 999–1010. IEEE.

[Wang et al., 2017] Wang, X., Zhu, S., Zhou, D., and Yang, Y. (2017). Droid-antirm: Taming control flow anti-analysis to support automated dynamic analysis of android malware. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 350–361.

[Wei et al., 2014] Wei, F., Roy, S., and Ou, X. (2014). Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341.

[Wei et al., 2016] Wei, L., Liu, Y., and Cheung, S.-C. (2016). Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237.

[Wei et al., 2019] Wei, L., Liu, Y., and Cheung, S.-C. (2019). Pivot: learning api-device correlations to facilitate android compatibility issue detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 878–888. IEEE.

[Wei et al., 2018] Wei, L., Liu, Y., Cheung, S.-C., Huang, H., Lu, X., and Liu, X. (2018). Understanding and detecting fragmentation-induced compatibility issues for android apps. *IEEE Transactions on Software Engineering*, 46(11):1176–1199.

[Wu et al., 2013] Wu, L., Grace, M., Zhou, Y., Wu, C., and Jiang, X. (2013). The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634.

[Xia et al., 2020] Xia, H., Zhang, Y., Zhou, Y., Chen, X., Wang, Y., Zhang, X., Cui, S., Hong, G., Zhang, X., Yang, M., et al. (2020). How android developers handle evolution-

induced api compatibility issues: a large-scale study. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 886–898. IEEE.

[Xiaoyu Sun, 2021] Xiaoyu Sun (2021). Improve FlowDroid to support Field Sources(develop branch) . `https://github.com/secure-software-engineering/FlowDroid/pull/385`. Online; accessed 12 September 2021.

[Xu et al., 2020] Xu, G., Hu, Y., Guo, Q., He, R., Li, L., Xu, G., Han, Z., and Wang, H. (2020). Dissecting mobile offerwall advertisements: An explorative study. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 518–526.

[Xu et al., 2022] Xu, G., Li, S., Zhou, H., Liu, S., Tang, Y., Li, L., Luo, X., Xiao, X., Xu, G., and Wang, H. (2022). Lie to me: Abusing the mobile content sharing service for fun and profit. In *Proceedings of the ACM Web Conference 2022*, pages 3327–3335.

[Xu et al., 2015] Xu, W., Li, J., Shu, J., Yang, W., Xie, T., Zhang, Y., and Gu, D. (2015). From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425.

[Xu et al., 2012] Xu, Z., Bai, K., and Zhu, S. (2012). Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 113–124.

[Yan et al., 2017] Yan, H., Sui, Y., Chen, S., and Xue, J. (2017). Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 42–54.

[Yang et al., 2017] Yang, X., Lo, D., Li, L., Xia, X., Bissyandé, T. F., and Klein, J. (2017). Characterizing malicious android apps by mining topic-specific data flow signatures. *Information and Software Technology*.

[Zeng et al., 2018] Zeng, Q., Luo, L., Qian, Z., Du, X., and Li, Z. (2018). Resilient decentralized android application repackaging detection using logic bombs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 50–61.

[Zhang et al., 2014] Zhang, M., Duan, Y., Yin, H., and Zhao, Z. (2014). Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1105–1116.

[Zhang et al., 2020] Zhang, Q., Wang, J., Gulzar, M. A., Padhye, R., and Kim, M. (2020). Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 722–733. IEEE.

[Zhang et al., 2019] Zhang, R., Chen, X., Wen, S., Zheng, X., and Ding, Y. (2019). Using ai to attack va: a stealthy spyware against voice assistances in smart phones. *IEEE Access*, 7:153542–153554.

[Zhang et al., 2015] Zhang, T., Gao, J., Cheng, J., and Uehara, T. (2015). Compatibility testing service for mobile applications. In *2015 IEEE Symposium on Service-Oriented System Engineering*, pages 179–186. IEEE.

[Zhang et al.,] Zhang, X., Wang, X., Slavin, R., and Niu, J. Condysta: Context-aware dynamic supplement to static taint analysis.

[Zhao et al., 2022] Zhao, Y., Li, L., Liu, K., and Grundy, J. (2022). Towards automatically repairing compatibility issues in published android apps. In *The 44th International Conference on Software Engineering (ICSE 2022)*.

[Zhao et al., 2021] Zhao, Y., Li, L., Wang, H., Cai, H., Bissyande, T., Klein, J., and Grundy, J. (2021). On the impact of sample duplication in machine learning based android malware detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*.

[Zheng et al., 2012] Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. (2012). Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104.

[Zhou and Walker, 2016] Zhou, J. and Walker, R. J. (2016). Api deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 266–277.

[Zhou et al., 2014] Zhou, X., Lee, Y., Zhang, N., Naveed, M., and Wang, X. (2014). The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423. IEEE.

[Zhu et al., 2013] Zhu, J., Wu, P., Wang, X., and Zhang, J. (2013). Sensec: Mobile security through passive sensing. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 1128–1133. IEEE.

[Zolfaghari et al., 2021] Zolfaghari, B., Parizi, R. M., Srivastava, G., and Hailemariam, Y. (2021). Root causing, detecting, and fixing flaky tests: state of the art and future roadmap. *Software: Practice and Experience*, 51(5):851–867.